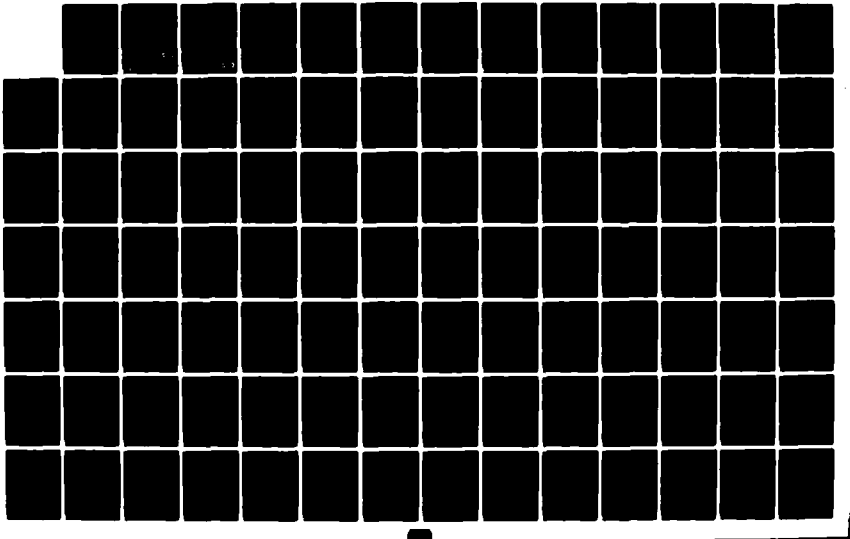
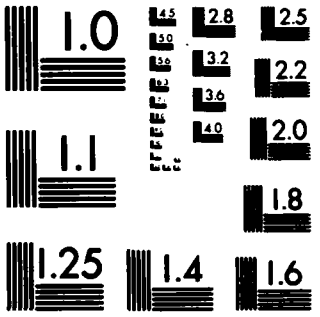
AD-A138 429 DESIGN AND IMPLEMENTATION OF AN INPUT/OUTPUT INTERFACE 1/4
PROTOCOL FOR THE I..(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. K N COLE
UNCLASSIFIED DEC 83 AFIT/GE/EE/83D-17 F/G 17/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A138429

DESIGN AND IMPLEMENTATION OF AN INPUT/OUTPUT INTERFACE PROTOCOL FOR THE INTEL 432/670 COMPUTER SYSTEM

THESIS

AFIT/GE/EE/83D-17 Kenneth N. Cole
1Lt USAF

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

DTIC
ELECTE
FEB 29 1984
S D
B

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DTIC FILE COPY

84 02 29 049

AFIT/GE/EE/83D-17

DESIGN AND IMPLEMENTATION OF AN INPUT/OUTPUT INTERFACE PROTOCOL FOR THE INTEL 432/670 COMPUTER SYSTEM

THESIS

AFIT/GE/EE/83D-17 Kenneth N. Cole
1Lt USAF

Approved for public release; distribution unlimited.

DTIC
ELECTE
FEB 29 1984
S D
B

AFIT/GE/EE/83D-17

DESIGN AND IMPLEMENTATION OF AN INPUT/OUTPUT INTERFACE PROTOCOL FOR THE INTEL 432/670 COMPUTER SYSTEM

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Electrical Engineering

by

Kenneth N. Cole, B.S., B.S.E.E.
First Lieutenant USAF
Graduate Electrical Engineering

December 1983

Approved for public release; distribution unlimited.

Preface

This thesis presents the design and development of an I/O Interface for a multiprocessor system, using a Computer Based Message System. This is only the beginning of the work to be done with the Intel 432 Computer System. The unique environment provided by the 432's architecture will entice systems designers to work towards realizing the system's full potential. I hope the I/O Interface system can provide a "stepping stone" towards development of greater things with the 432 system.

I would like to thank my thesis advisor, Dr. Gary B. Lamont, for his guidance and, especially, for the freedom he gave me in this work. I must, also, thank Lt. Col. Hal Carter for his simple nods, as I tried to explain what I was doing. They caused me to re-examine my work, more than once.

Most of all, I would like to express my love and appreciation to my wife, Mary Anne, and my daughter, Christina. Their patience and understanding made this possible. What can I say to someone who smiles, when I tell her that I won't take anymore classes?

Kenneth N. Cole

Table of Contents

	Page
Preface	ii
List of Figures	vi
List of Tables	xi
Abstract	xiv
Glossary of Terms	xv
I. INTRODUCTION	1-1
Background	1-1
Problem Statement	1-7
Scope and Limitations	1-7
General Approach	1-9
Equipment	1-11
Sequence of Presentation	1-11
II. REQUIREMENTS	2-1
Introduction	2-1
Conceptual Requirements of the I/O Interface Protocol	2-1
Virtual Operation	2-2
Flexibility	2-3
Conceptual Standards	2-4
Functional Requirements for the I/O Interface Protocol	2-7
Relation to the System	2-8
Model of Operation	2-8
Functional Description	2-11
Address Mechanism	2-12
Format Mechanism	2-13
Control Mechanism	2-14
Reply Mechanism	2-15
Functional Standards	2-16
User Address	2-17
Message Format	2-17
Implementation Constraints	2-20
Hardware	2-20
432 Micromainframe	2-22
432 Cross Development System	2-23
Software	2-25
432 Software	2-25
Attached Processor Software	2-26
Summary	2-27
III. SYSTEM DESIGN	3-1
Introduction	3-1
General Design Features	3-2
System Processes	3-2
Virtual Devices	3-3
System Flexibility	3-4

Table of Contents (continued)

		Page
	Functional Mechanism Implementation	3-6
	Address Mechanism	3-6
	Format Mechanism	3-7
	Device Control Mechanism	3-8
	Device Reply Mechanism	3-8
	User Sublayer	3-8
	User Shell	3-9
	Shell	3-10
	User Interface	3-11
	System Commands	3-12
	Device Abstraction	3-15
	User Agent Sublayer	3-19
	User Shell Agent	3-23
	Device Agents	3-25
	Message Transfer Sublayer	3-27
	Message Transfer System	3-29
	Address Manager	3-31
	Route Manager	3-34
	CBMS Manager	3-35
	Summary	3-36
IV.	SYSTEM TESTING	4-1
	Introduction	4-1
	Environment Validation Test	4-5
	432 Processor System Validation	4-8
	User Sublayer Test (432)	4-9
	User Agent Sublayer Test (432)	4-11
	Message Transfer Sublayer Test (432)	4-13
	Attached Processor System Validation	4-15
	User Sublayer Test (AP)	4-17
	User Agent Sublayer Test (AP)	4-18
	Message Transfer Sublayer Test (AP)	4-20
	System Integration Test	4-21
	Summary	4-22
V.	RESULTS, CONCLUSIONS AND RECOMMENDATIONS	5-1
	Introduction	5-1
	Test Results	5-1
	Environment Validation Test	5-1
	432 Processor Tests	5-2
	Attached Processor Tests	5-3
	Conclusions	5-3
	Recommendations for Future Study	5-6
	Inter-Process Communications Projects	5-7
	Intel 432/670 Computer System Projects	5-8

Table of Contents (continued)

	Page
Bibliography	BIB-1
Appendix A: Intel 432/670 System Architecture	A-1
Appendix B: Object Oriented Systems Design	B-1
Appendix C: iMAX 432 Multifunction Applications Executive	C-1
Appendix D: I/O Interface Message Format	D-1
Appendix E: DELNET Network Addressing Scheme	E-1
Appendix F: Intel 432 Cross Development System Hardware and Software Compatibility Guide	F-1
Appendix G: Ada Compiler Unimplemented Facilities . .	G-1
Appendix H: I/O Interface User's Manual	H-1
Appendix I: I/O Interface Data Flow Diagrams	I-1
Appendix J: I/O Interface Software Structure Charts .	J-1
Appendix K: Test Results	K-1
Vita	V

Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

DTIC COPY INSPECTED 2

List of Figures

Figure		Page
1-1	Intel 432/670 System Processors	1-2
1-2	AFIT/ENG 432/670 System I/O Hardware Organization	1-3
1-3	ISO Reference Model Organization	1-4
1-4	Logical Model of a Computer Based Message System	1-5
2-1	ISO Open Systems Interconnection Reference Model	2-4
2-2	Message Transfer Protocols in the ISO Reference Model	2-5
2-3	CBMS within the I/O Interface Protocol . .	2-6
2-4	I/O Interface Protocol Relationship to the Other Protocols	2-7
2-5	Model of I/O Interface Protocol Organization within the ISO Reference Model	2-9
2-6	CBMS Message Structure	2-19
2-7	Intel 432/670 Micromainframe Hardware Configuration	2-21
2-8	The Intel 432 Cross Development System . .	2-23
3-1	I/O Interface Context Diagram	3-5
3-2	I/O Interface Sublayers within the ISO Applications Layer	3-6
3-3	User Shell Context Diagram	3-9
3-4	User Shell Main Program Data Flow	3-10
3-5	Perform_System_Command Data Flow	3-14
3-6	Typical User Agent Receive Procedure Data Flow	3-19
3-7	Typical User Agent Send Procedure Data Flow	3-21

List of Figures (continued)

Figure		Page
3-8	Typical User Agent Initialization Data Flow	3-22
3-9	I/O Interface System Organization	3-28
3-10	Mts_Receive Data Flow	3-30
3-11	Mts_Send Data Flow	3-30
3-12	I/O Interface Device Naming Structure . . .	3-33
4-1	I/O Interface Software System Structure . .	4-2
4-2	Environment Validation Test Hardware Configuration	4-7
4-3	Attached Processor Validation Test Hardware Configuration	4-15
A-1	iAPX 432 Minimum System Organization . . .	A-3
A-2	Two-Level Mapping	A-4
A-3	GDP Program Structure	A-9
A-4	IP Program Structure	A-11
A-5	General 432/670 System Configuration . . .	A-16
A-6	Intel 432/670 Micromainframe Multiprocessor Configuration	A-17
B-1	A Module in the Object-Oriented Methodology	B-3
C-1	Static Process State Transitions	C-2
C-2	Dynamic Process State Transitions with BPM	C-6
C-3	iMAX Storage Management Transitions	C-9
C-4	iMAX Message AD State Transitions	C-11
C-5	432/670 System Bus Hardware Configuration .	C-18
D-1	General Message Structure	D-2

List of Figures (continued)

Figure		Page
D-2	Encoding Mechanism for Qualifiers and Length Codes	D-4
E-1	I/O Interface Message Address Format . . .	E-2
H-1	Intel 432 Cross Development System Hardware Environment	H-4
H-2	System 432/670 Standard Configuration . . .	H-6
H-3	432/670 Cross Development System Hardware Configuration	H-8
H-4	AFIT/ENG 432/670 Computer System Hardware Configuration	H-13
H-5	AFIT/ENG 432/670 I/O Interface System . . .	H-19
H-6	User View of the I/O Interface System . . .	H-21
H-7	User Agent Receive Process Data Flow . . .	H-22
H-8	Typical User Agent Send Procedure Data Flow	H-24
H-9	User Shell Operating Hardware Configuration	H-36
H-10	Help Command Syntax	H-40
H-11	Set Command Syntax	H-43
H-12	I/O Interface Device Naming Structure . . .	H-44
H-13	Copy Command Syntax	H-45
J-1	432 System Initialization Structure Chart (0.1)	J-3
J-2	Attached Processor System Initialization Structure Chart (0.1)	J-4
J-3	Main (User Shell) Structure Chart (0.2) . .	J-5
J-4	Perform System Command (System Commands) Structure Chart (2.1)	J-6

List of Figures (continued)

Figure		Page
J-5	Determine System Command (System Commands) Structure Chart (2.1.1)	J-7
J-6	Set (System Commands) Structure Chart (2.1.1.1)	J-8
J-7	Help (System Commands) Structure Chart (2.1.1.2)	J-9
J-8	Copy (System Commands) Structure Chart (2.1.1.3)	J-10
J-9	USA_Open (User Shell Agent) Structure Chart (6.2)	J-11
J-10	USA_Close (User Shell Agent) Structure Chart (6.3)	J-12
J-11	USA_Read (User Shell Agent) Structure Chart (6.4)	J-13
J-12	USA_Write (User Shell Agent) Structure Chart (6.5)	J-14
J-13	USA_Page (User Shell Agent) Structure Chart (6.6)	J-15
J-14	USA_Title (User Shell Agent) Structure Chart (6.7)	J-16
J-15	USA_Delete (User Shell Agent) Structure Chart (6.8)	J-17
J-16	USA_Rename (User Shell Agent) Structure Chart (6.9)	J-18
J-17	USA_Reset (User Shell Agent) Structure Chart (6.10)	J-19
J-18	USA_Get_Config (User Shell Agent) Structure Chart (6.11)	J-20
J-19	USA_Set_Config (User Shell Agent) Structure Chart (6.12)	J-21
J-20	USA_Test (User Shell Agent) Structure Chart (6.13)	J-22

List of Figures (continued)

Figure		Page
J-21	USA_Receive (User Shell Agent) Structure Chart (6.1.1)	J-23
J-22	PSA_Receive (Printer System Agent) Structure Chart (7.1.1)	J-24
J-23	IFSA_Receive (ISIS File System Agent) Structure Chart (8.1.1)	J-25
J-24	S3CA_Receive (Series III Console Agent) Structure Chart (9.1.1)	J-26
J-25	MTS_Receive (Message Transfer System) Structure Chart (10.1.1)	J-27

List of Tables

Table		Page
2-I	I/O Interface Message Fields	2-18
2-II	Summary of I/O Interface Requirements . . .	2-28
3-I	I/O Interface Functions and Reply Codes . .	3-3
3-II	Procedures of the User Interface Package . .	3-11
3-III	Procedures of the System Commands Package .	3-13
3-IV	Procedures of the Printer System Package . .	3-16
3-V	Procedures of the ISIS File System Package .	3-17
3-VI	Procedures of the Series III Console Package	3-18
3-VII	Procedures of the User Shell Agent Package .	3-23
3-VIII	I/O Interface Replies to Function Requests .	3-24
3-IX	Procedures of the Device Agent Packages . .	3-25
3-X	I/O Interface Function Mapping to System Devices	3-26
3-XI	Procedures of the Message Transfer System Package	3-29
3-XII	Procedures of the Address Manager Package .	3-31
3-XIII	Procedures of the Route Manager Package . .	3-34
3-XIV	Procedures of the CBMS Manager Package . . .	3-35
4-I	I/O Interface Testing Procedures	4-4
4-II	Environment Verification Test Procedure . .	4-5
4-III	PRIME Program Software	4-8
4-IV	432 Processor Software Validation Test Command List	4-9
4-V	User Sublayer Validation Software (432) . .	4-10
4-VI	User Agent Sublayer Validation Software (432)	4-12

List of Tables (continued)

Table		Page
4-VII	Message Transfer Sublayer Validation Software (432)	4-13
4-VIII	Attached Processor Test Shell Commands . . .	4-16
4-IX	User Sublayer Validation Software (AP) . . .	4-17
4-X	User Agent Sublayer Validation Software (AP)	4-19
4-XI	Message Transfer Sublayer Validation Software (AP)	4-20
4-XII	System Integration Test Commands	4-21
5-I	Summary of Test Results	5-2
5-II	Summary of Recommendations for Future Study	5-7
C-I	Comparison of Ada Tasks and iMAX BPM Processes	C-3
C-II	iMAX 432 Storage Management Capabilities . .	C-8
C-III	Syncronous I/O Interface Operations and Device Types	C-14
C-IV	Asyncronous Interface Command and Reply Cross-Reference	C-16
D-I	I/O Message Fields	D-3
D-II	I/O Interface Device Names	D-11
D-III	CBMS Address Field Values	D-12
F-I	Hardware and Software Compatibility Guide .	F-2
G-I	Ada Compiler System Implementation Restrictions	G-4
H-I	432 Cross Development System VAX/VMS Directories	H-9
H-II	432 Cross Development System Series III Workstation Software	H-11

List of Tables (continued)

Table		Page
H-III	I/O Interface 432 Software Packages	H-16
H-IV	I/O Interface 8086 Software Packages	H-18
H-V	I/O Interface Replies to Function Requests .	H-23
H-VI	Mapping of I/O Interface Commands to Device Functions	H-26
H-VII	Printer System Functions	H-28
H-VIII	ISIS File System Functions	H-29
H-IX	Series III Console Device Functions	H-31
H-X	Guidelines for Adding New Devices to the I/O Interface	H-33
H-XI	User Shell System Files	H-37
H-XII	Logical Operands for Command Syntax Definition	H-39
H-XIII	Help Command Response (Default)	H-41
H-XIV	Help Command Response (Set Command Query) .	H-41
H-XV	Help Command Response (Help Command Query) .	H-42
H-XVI	Help Command Response (Copy Command Query) .	H-42

Abstract

Distributed computer systems have many advantages to offer in terms of simplicity, efficiency, protection, and security as well as improved performance from the concurrency in such a system. Communication among distributed processors is a key issue in the design of a distributed system. While the lower levels of a communication system are generally defined by the hardware configuration and thus, implementation dependent, protocols for communication may be developed at higher levels that are independent of the hardware implementation. Using the Computer Based Message System under development for the National Bureau of Standards, this thesis investigation is an attempt to develop a usable I/O interface for a distributed computer system.

The Intel 432 Micromainframe computer system is a functionally distributed multiprocessor system. The hardware organization and operating system features lend themselves to the implementation of a message based communication system among users and devices on distinct processor.

This specific research effort involves defining the protocol requirements, as well as designing, implementing and testing a distributed I/O system communication interface on the 432 computer system.

Glossary

ACS: Ada Compiler System.

Ada: A registered trademark of the United States Department of Defense, Under Secretary for Research and Engineering. The programming language defined by the document ANSI/MIL-STD-1815A, dated 22 January 1983.

Ada Compiler System: The Intel Ada language cross compiler which executes on the VAX-11/780 computer system and produces object code for the iAPX 432 processor hardware.

Address Manager: The software package containing procedures which define the mapping of I/O Interface device names to their unique CBMS addresses.

Address Mechanism: The method of uniquely designating entities of the system which may be source or destination points for messages.

AFIT: Air Force Institute of Technology, at Wright-Patterson A.F.B, Ohio.

AM: Address Manager.

AP: Attached Processor.

Applications Layer: The seventh (highest) layer of the ISO Reference Model for Open Systems Interconnection.

ASM-86: An assembler for the 8086 microprocessor. Produces machine executable code from mnemonic assembly language instructions.

Attached Processor: A processor element of the Intel 432/670 Micromainframe Computer System. A computer system containing an Interface Processor board which is connected to the Interface Processor Link board on the system bus of the 432/670 chassis.

CBMS: Computer Based Message System.

CBMS Address: A unique 32-bit designation for the device abstraction or user process.

Complete Device Name: An I/O Interface User Shell device name containing identifiers for all four parts; country code, network code, host code, and user-id.

Computer Based Message System: A system of protocols being developed by the National Bureau of Standards, Institute for Computer Sciences and Technology.

CON: The I/O Interface User Shell device name for the Series III MDS console using the ISIS operating system.

Control Mechanism: The method of causing the system devices to perform specific actions.

Country-Code: The designation for the most significant part of the DELNET network user naming scheme.

Cross Compiler: A language compiler program that executes on one computer system and produces machine code for another computer.

Data Element: A part of a field (in a message). A data element contains four parts: the identifier, the length, the qualifier, and the data contents.

Data Flow Diagram: A schematic representation of the information movement within a portion of a computer program.

Data Link: The second layer of the ISO Reference Model for Open Systems Interconnection.

Dead Lock: The state of a system when further processing is not possible, due to interaction between concurrent processes. (e.g., each of two processes has control of an asset that the other is waiting for).

Debugger: The designation for the Intel Series III MDS which is an Attached Processor to the Intel 432/670 system and is executing the DEB432 software to act as the software debugging workstation during 432 software development.

Debug Workstation: The console of the Debugger system (Intel Series III MDS).

DELNET: Digital Electronics Laboratory NETwork.

Device Abstraction: The logical definition of a device by description of its functions. Also, a set of functions which define the perception of a device.

DFD: Data Flow Diagram.

Digital Engineering Laboratory Network: A network system, currently under development at AFIT, that includes system nodes implementing the lower three levels of the ISO Reference Model.

DOD: Department of Defense. A bureaucratic organization of the United States Government.

DSK: The I/O Interface User Shell device name for the Series III MDS disk system using the ISIS operating system.

Dummy Module: A procedure used during testing, to simulate a procedure that has not been coded. In general, dummy procedures perform only those actions essential to let the execution of the test code continue.

Field: A part of a message. A field consists of one or more data elements.

Flexibility: The ability of a system design to be gracefully modified to incorporate new elements.

Format Mechanism: The definition of organization for the message data structure.

GDP: General Data Processor.

General Data Processor: A processor element of the Intel 432/670 Micromainframe Computer System. A 2-chip microprocessor in the iAPX 432 microprocessor hardware family.

Host-Code: The third part of the DELNET network user naming scheme.

iAPX: Intel Advanced Processor System.

iAPX 432: A family of microprocessor elements, designed and manufactured by the Intel Corporation, of Santa Clara, California.

IDA: The command name, used in Intel documentation, for the Ada language compiler of the ACS.

IFS: ISIS File System.

IFSA: ISIS File System Agent.

iMAX: Intel Multifunction Applications Executive.

Interface Processor: A processor in the Intel 432/670 Micromainframe Computer System that acts as a communication system between the system bus of the 432/670 and the Attached Processor system.

I/O: Input and Output.

IP: Interface Processor.

IPC-85: The Integrated Processor Card for the Intel Series III MDS system, based on the 8085 microprocessor.

ISIS: Intel System Implementation Supervisor. The operating system designed for the Intel Development Systems.

ISIS File System: The I/O Interface device abstraction providing access to the disk file system of the Intel Series III MDS using the ISIS operating system facilities.

ISIS File System Agent: The CBMS User Agent for the ISIS File System entity.

ISO: International Standards Organization.

LINK432: The command name of the linker program of the Intel 432 Cross Development System.

LINK-86: A linker program for the 8086 microprocessor which creates an executable code module from several related object code modules.

MDS: Microcomputer Development System. Generally used in connection with the Intel Series III Microcomputer Development System.

Message: A string of bytes (8-bit data) representing data to be communicated from one entity to another.

Message Transfer System: The CBMS entity which is responsible for moving messages between User Agents.

Message Transfer Sublayer: The first (lowest) layer of the CBMS model. A sublayer of the Applications layer of the ISO model.

Micromainframe: A registered trademark of the Intel Corporation. Refers to mainframe-like features of the microprocessor based 432/670 system architecture.

MULTIBUS: A registered trademark of the Intel Corporation, referring to the bus structure defined by the IEEE 796 standard.

MTS: Message Transfer System.

NBS: National Bureau of Standards.

NET0: DELNET Network Code designation for the original network containing the Intel system.

Network-Code: The second part of the DELNET network user naming scheme.

Network Layer: The third layer of the ISO Reference Model for Open Systems Interconnection.

Object-Oriented: A system design methodology that is focused on using the implementation of system structures as the basic entity for building complex organizations. Generally contrasted with methodologies based upon the implementation of system functions.

PL/M-86: A high-order language for the 8086 microprocessor. Similar in organization to the PL/I language.

Physical Layer: The first (lowest) layer of the ISO Reference Model for Open Systems Interconnection.

Port: In hardware or software, a point of communications interface where a device or process expects to receive information.

Port-Code: The fourth (and least significant) part of the DELNET network user naming scheme.

Pragma: A compiler command for the Ada language.

Presentation Layer: The sixth layer of the ISO Reference Model for Open Systems Interconnection.

PRIME: Program name for the prime number computing program, an example Ada language program, supplied by Intel, for the 432/670 system.

Printer System: The I/O Interface device abstraction providing access to the TTY serial port of the Intel Series III MDS through the ISIS operating system.

Printer System Agent: The CBMS User Agent for the Printer System.

Property List: A field within the CBMS message structure. The Property List field is not used in the I/O Interface message structure.

Protocol: The definition of a systematic structure of communication.

PRT: The I/O Interface User Shell device name for the Printer System device.

PS: Printer System.

PSA: Printer System Agent.

Remote System: The outer-most system in a star network of computer systems or the Attached Processor system of the Intel 432/670 system organization.

Reply Code: An 8-bit element of the Reply Message set by the User Agent of the destination device and returned, by the CBMS system, to the source user process.

Reply Mechanism: The method of providing device status information in response to a request for action.

RM: Route Manager.

RM67: DELNET Country Code designation for Room 67 of Building 640 where the Intel systems are located.

RPB-86: The Remote Processor Board of the Intel Series III MDS. A special configuration of the Intel iSBC 86/12A processor board.

Route Manager: A software package containing procedures which define the mapping of CBMS addresses to the ports where the user, or device, expects to receive messages.

Series III Console: The I/O Interface device abstraction providing access to the console of the Intel Series III MDS Attached Processor using the ISIS operating system facilities.

Series III Console Agent: The CBMS User Agent for the Series III Console entity.

Session Layer: The fifth layer of the ISO Reference Model for Open Systems Interconnection.

Static Task: A task that is started at the time of system initialization and cannot be destroyed.

S3C: Series III Console.

S3CA: Series III Console Agent.

Task: A single process entity within a computer program.

Top-Down: The sequence of system development beginning with the most general (highest) concepts and ending with the most specific (lowest) features.

Transport Layer: The fourth layer of the ISO Reference Model for Open Systems Interconnection.

UA: User Agent.

UNID: the Universal Network Interface Device. The network node hardware device of the DELNET system.

US: User Shell.

USA: User Shell Agent.

User Agent: The CBMS entity which provides access to the message system for the using process.

User Agent Sublayer: The second layer of the CBMS model. A sublayer of the Applications layer of the ISO model.

User-Id: The least significant part of the I/O Interface device name. Also known as the Port-Code of the DELNET naming scheme.

User Shell: The using process developed to demonstrate the I/O Interface.

User Sublayer: The third (highest) layer of the CBMS model. A sublayer of the Applications layer of the ISO model.

USR: The I/O Interface User Shell device name for the User Shell process (i.e., the Series III MDS Debugger console).

Virtual Machine: A set of functions which define the apparent operation of a system.

I. INTRODUCTION

Background

Since the development of the digital computer, researchers in government and private industry have been trying to improve the capabilities of their systems. Computers of smaller size and greater power are now available to the general public. The Intel iAPX 432 family is an extension of the current state of these developments.

The Intel 432/670 Micromainframe Computer System is a multiprocessor system containing three distinct processor types; General Data Processors (GDPs), Interface Processors (IPs), and Attached Processors (APs). The physical relationship between these three processors is shown in Figure 1-1. The system architecture forms a network of processors. Within this network, the GDPs are the central processors of the system; the APs are the input/output (I/O) processors for the system, and the IPs provide communication facilities between the GDPs and APs. All system I/O is handled by the APs, which are the external nodes of the network arrangement. Appendix A, included with this report, gives a more detailed description of the 432/670 System.

The Attached Processors may be any 8/16-bit processor system that is capable of MULTIBUS connection with the Interface Processor (see Appendix A). Different processors may be used to add nodes with a wide range of intelligence.

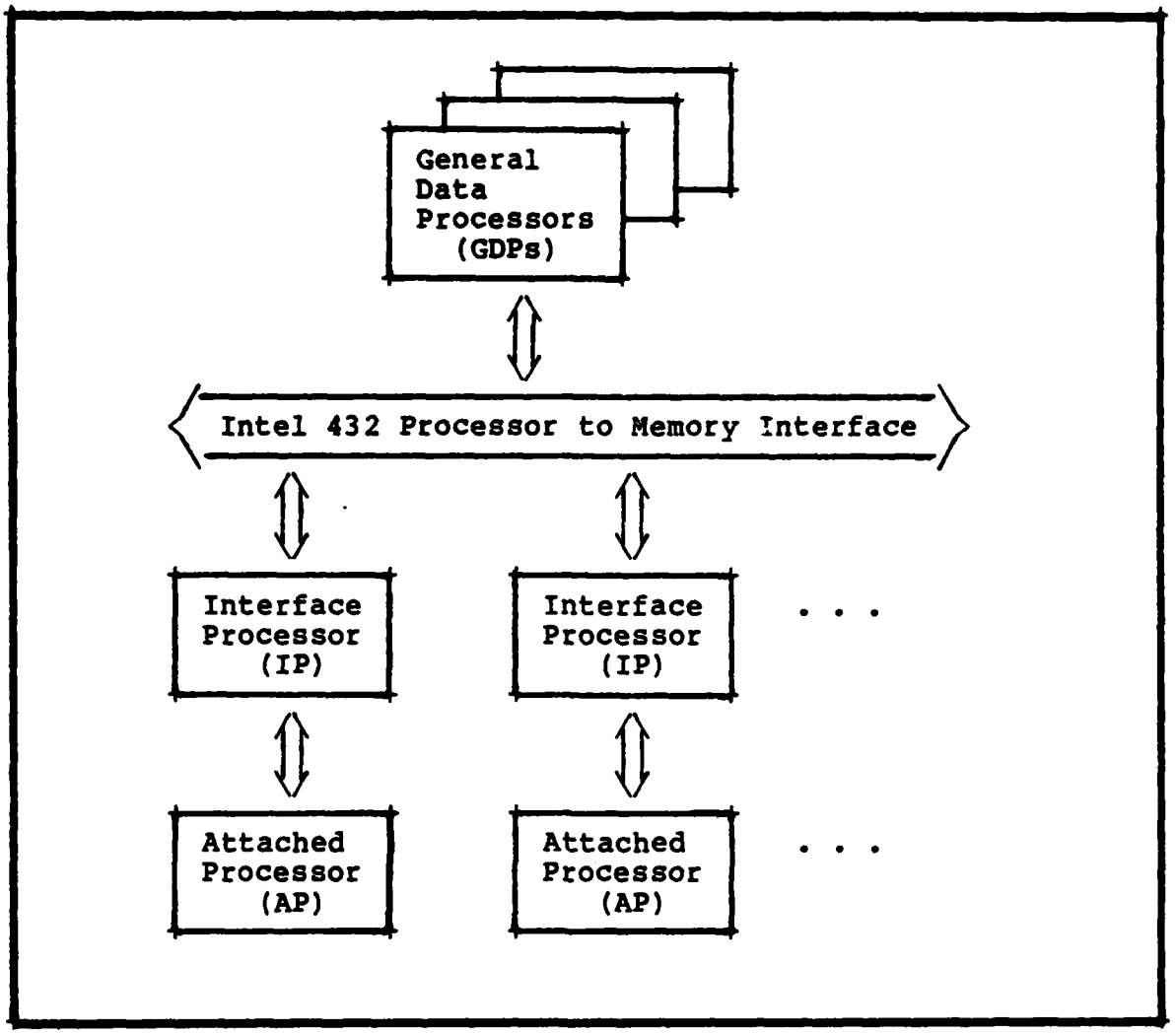


Figure 1-1. Intel 432/670 System Processors

One approach to providing I/O facilities for the 432 system, would be use of a commercial computer system as an AP. An appropriate system would be the Intel Series III Microcomputer Development System (MDS) which has a compatible bus structure and been designed for development of hardware and software systems.

The Series III MDS has a number of peripheral devices

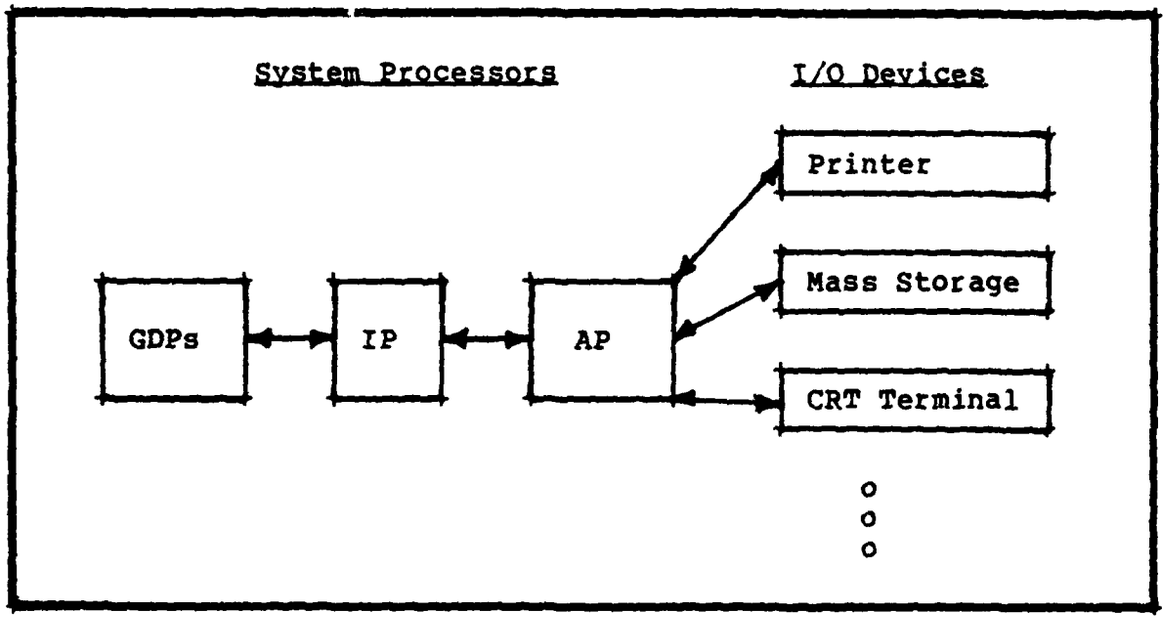


Figure 1-2. AFIT/ENG 432/670 System
I/O Hardware Organization

which may be accessed through its operating system (ISIS) driver routines. Software can be developed on this system to allow the 432 processor to access the existing I/O facilities of the Series III MDS through the IP. Figure 1-2 shows this basic organization of hardware. This type of resource sharing is one of the primary functions of computer networks (Ref 31:1-2). It follows, then, that the software structure, of the 432 system I/O interface software, should follow the general software organization for computer network systems, in order to effectively apply the concepts of network organization to the 432 computer system.

The International Standards Organization (ISO) has developed a Reference Model for Open Systems Interconnection

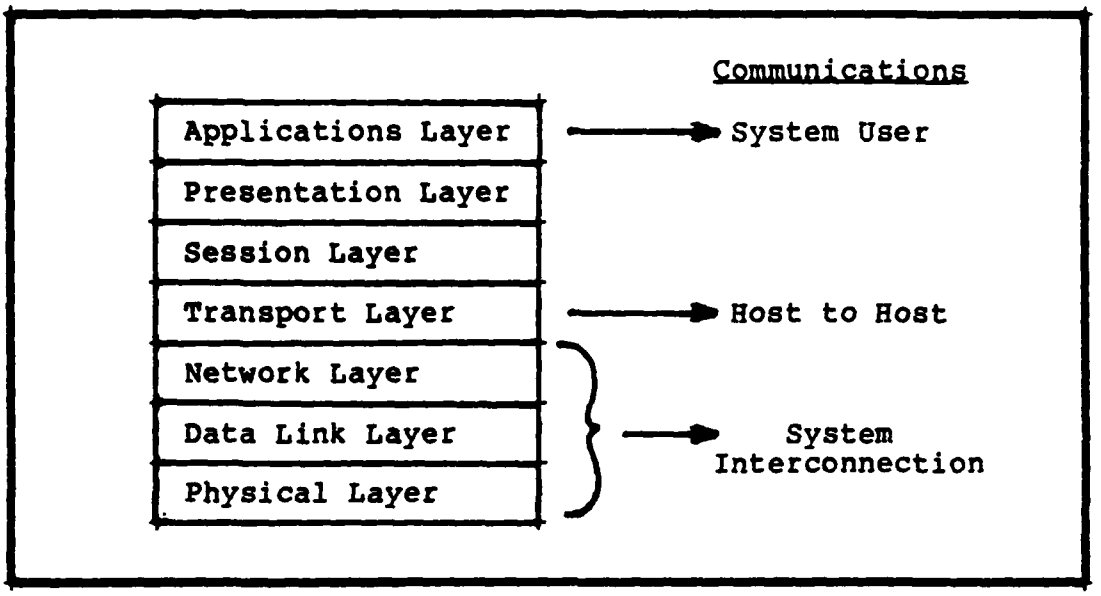


Figure 1-3. ISO Reference Model Organization

which has been used as a standard for network system functional organization (Ref 11,30,32). Figure 1-3 shows the hierarchy of layers in the ISO model. The lowest three layers (Physical, Data Link, and Network) establish the interconnection of network nodes. The Transport layer uses the interconnection facilities of the Network layer to provide reliable host-to-host communication facilities to the layers above. The protocol of the Transport layer defines the mechanism for message based communications between distinct processors of a network system. The Session and Presentation layers are, often, grouped with the Applications layer, as providing services only required for specific user functions. The Presentation layer provides data manipulations necessary for interpretation of the

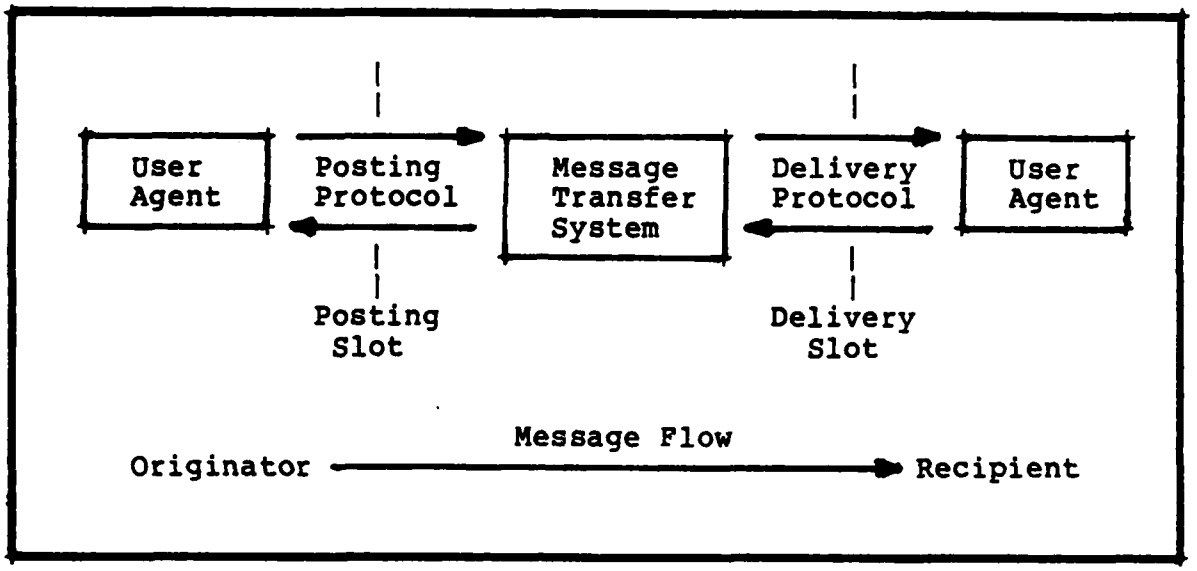


Figure 1-4. Logical Model of a Computer Based Message System (Ref 8:3)

information (e.g., coding or character set transformations). The Session layer provides for process to process communication, generally, of longer duration that a simple message and response system. A minimum communications system does not require these services. The tasks that use the communications system, then, are placed in the highest layer (Applications layer).

One such application of the communication facilities is a message transfer system. The U.S. National Bureau of Standards is working to develop Computer Based Message System (CBMS) as a part of a family of computer network protocols (Ref 8:1-5). Figure 1-4 shows the logical model of the CBMS. The "User Agent" is a functional entity, that acts on behalf of a user, helping to create and handle

messages and communicate with the Message Transfer System. The Message Transfer System (MTS) is a process that accepts messages from an originating User Agent and passes them to the receiving User Agent. The MTS may use the communication facilities of the lower layers to move messages from one computer system to another (Ref 8:2-3).

There is a similarity of structure between the CBMS organization, shown in Figure 1-4, and the physical interconnection of the processors of the 432/670 computer system. The ISO and NBS standards define an organization of users which are interconnected by lines of communication. The architecture of the 432 system is a network of processors (432 central processor and multiple APs) where there is a need for communication between system users on the 432 processor and I/O devices on the APs. This suggests that the development of an interface protocol for I/O for this configuration. Using the CBMS satisfies the need for an I/O communication system and meets the requirements of the ISO Reference Models structure. Implementation of the protocol on the AFIT/ENG 432/670 Computer System allows future users of the system to build on this interface. This protocol should also provide access to the I/O systems of the Series III Microcomputer Development System (a 432 system AP).

Problem Statement

The purpose of this study is to design, implement and test a message-based input/output interface for the Intel 432/670 Micromainframe Computer System, which provides access to the I/O devices of the Intel Series III Microcomputer Development System.

Scope and Limitations

The Series III Microcomputer Development System is used as the I/O processor system because it is an available system which uses the MULTIBUS (IEEE Standard 796) structured system bus (see Intel System 432/600 System Reference Manual (Ref 24) for system hardware interface requirements). The Series III MDS also provides a software development environment which is compatible with the 432 operating system modules that must execute in the AP (see Appendix C). Other MULTIBUS-based systems (Ref 18,19) are not available at the AFIT Digital Engineering lab or cannot provide an established software development environment with I/O devices already implemented.

The design of the I/O Interface protocol includes the specification of device abstractions which define each I/O device as a set of operations. Also, the functions available to users of the interface and the mapping of those functions onto the operations of each I/O device must be defined. These definitions create a logical specification of the I/O Interface.

The system design follows the hierarchial structure of the ISO Reference Model and the NBS Computer Based Message System to maintain agreement with accepted standards and the DELNET system (Ref 32). Structure charts provide for all I/O Interface software and test procedures allow for validation of each level of the interface program structure.

An I/O Interface Users Guide is provided (Appendix H), which includes instructions on the use and operation of the interface. Also, a procedure is given for implementation of new devices into the I/O Interface system. The Users Guide is intended to be a complete reference for use of the AFIT/ENG 432/670 Computer System implementation of the I/O Interface.

This implementation provides for only three devices; the Printer System, the Series III MDS console, and the ISIS File System. These were selected as the devices most often connected to the Series III MDS. Therefore, the devices most likely to be required for 432 system I/O operations. However, the common device access structure presented by the ISIS operating system allows any system I/O device to be used in a similar manner (Ref 13:1-1/1-5). Alternatives to using the ISIS device drivers would include using another operation system which provides device drivers or designing special device drivers within iRMX 86 operating system structure. These approaches may be used to make improvements to the system.

The device driver routines of the ISIS operating system are used by the I/O Interface software to simplify the implementation. Also, the 432 operating system (iMAX) functions to create and handle messages and communication ports are used by the lower level modules of the I/O Interface implementation. This is a divergence from the ISO Reference Model which specifies the services provided by the Transport Layer. This decision does not disrupt the structure of the entire design because the use of these procedures is restricted to the Message Transfer System modules which may be modified to use proper Transport Layer routines. However, the implementation of the Transport Layer on the 432 system is not necessary for validation of the I/O Interface Protocol design or use of the interface system.

General Approach

The initial approach is to perform a literature search for information on the design of related systems. Familiarization with the Intel 432 Micromainframe system is emphasized. Past AFIT theses and Intel publications on the 432 computer system is the starting point for this study.

It is the objective of this investigation to create an interface system which may be used in future development work with the 432. To this end, system flexibility and complete documentation is emphasized. The interface

protocol is designed to be practical for communication with the Series III MDS and ISIS operating system functions for purposes of this study. In addition, considerations are made for expansion of the protocol to handle future system needs. Users operating instructions are provided in Appendix H for both the I/O Interface and the user interface demonstrating the implementation.

The top-down design philosophy (Ref 38:32,50-176) is followed in the software development to create a maintainable system for future users. Also, Object-oriented design techniques, as described in Appendix B, are used to create a software organization consistent with the hardware and software architectures of the 432 (Appendix A) and its operating system, iMAX 432 (Appendix C).

Testing procedures are also emphasized to provide a secure foundation upon which new 432 System software projects may be developed. The testing procedures documented with this study (Chapter IV) are designed to be usable by future system implementors to test hardware and software configurations of the 432 System. These tests are used throughout the stepwise development of system software. The first test is validation of the 432 Development System environment. Next, validation tests are performed on the 432 processor and the AP, separately, to validate each system's hierarchy of software. Finally, the system is tested in a complete configuration. All the testing

procedures are documented with sufficient detail to enable the procedures to be duplicated.

Equipment

The hardware and software used in this thesis effort are listed in Appendix H of this report. It is available in the Digital Engineering Lab in building 640 of the Air Force Institute of Technology (AFIT).

Intel Corporation is working to improve the 432/670 System. Updates to the software and hardware of the system are provided to AFIT as they become available and future users of the 432 will probably find the development environment much improved over the current release of the system. However, to enable this thesis effort to progress smoothly, the software and hardware systems are "frozen" to the configuration described in the I/O Interface Users Guide (Appendix H).

Sequence of Presentation

This chapter presents an introduction to the work to be accomplished. Chapter II defines the requirements for the I/O Interface Protocol and system implementation. The system design and implementation are presented in Chapter III and the testing procedures are described in Chapter IV. Finally, the test results, conclusions and recommendations are given in Chapter V.

II. REQUIREMENTS

Introduction

The purpose of this chapter is to define the requirements for the I/O Interface Protocol for message based communication within a multiprocessor computer system and the implementation of the interface on the Intel 432/670 Micromainframe computer system (432/670). The requirements for the I/O Interface are presented in three parts. The conceptual requirements for the I/O Interface are described, first. Next, the functional requirements are defined independent of the implementation environment. The standards which apply are presented at the end of each section. Finally, the implementation constraints are described for the hardware and software environment of the AFIT/ENG 432/670 computer system.

Conceptual Requirements of the I/O Interface Protocol

The conceptual requirements deal with the characteristics of an ideal system. The input/output functions of a system are generally device specific procedures which are provided by the operating system to enable the user to perform I/O functions with simple commands (Ref 4:2-3). One possible idealization of an I/O device is a functional view where the device is defined by the user services. For example, an ideal printer might

consist of a set of printer-type functions (i.e., print_file, print_character, form_feed, etc.) which perform actions a user would require. The details of how the printer performs the functions are hidden from the user. The interface, with the printer functions, is free from dependency on the characteristics of a particular device. This virtual operation concept can be extended to the level above the devices. If all system devices respond to the same set of functions, then this common interface appears to the user as a simple set of general I/O functions. Virtual operation is thus, one requirement of the I/O Interface.

An operational system does not, generally, remain fixed. As the needs of the users change, the system is modified to meet those needs. The I/O configuration of the system may change significantly as new devices are added and old devices are modified or removed. The interface must be able to accept changes gracefully and not hinder the efforts to improve the system. Therefore, flexibility is a requirement of the I/O Interface.

Virtual Operation. Virtual operation implies that the user can communicate with any peripheral device in the system at the same level and in the same manner. For example, the user may read a line from a file structured device as easily as the keyboard input from a terminal. The characteristic differences of the operations, which depend on the particular devices, are hidden from the user. While,

servicing each request, on a various system devices, may require greatly different functions, the user issues a command that differs only in the device name. This transparency is essential to realizing efficiency and effectiveness in the I/O interface. Tradeoffs may be made to develop workable implementations within the limitations of the system environment discussed later in this chapter.

Flexibility. Flexibility in system design means planning for unforseeable events. The increasingly high costs of system design and development have forced engineers to consider the life-cycle of new designs more closely. New systems must be maintainable in their present condition and be able to grow with the needs of the user. It has become necessary to build in "flexibility" so that a product can be used for a variety of tasks with little or no modification. On the other hand, care must be taken not to over-design a new system or it will become too awkward and unwieldy for the user. Too comprehensive a system may not function efficiently in any application. Therefore, the application of "flexibility" is more art than science.

The I/O interface protocol must fulfill the needs of the system with regard to existing or predictable device interconnections and be flexible enough to allow for expansion and reconfiguration. While any new system's I/O needs are minimal at first, the protocol must be capable of handling peripheral devices whose structure or use is not

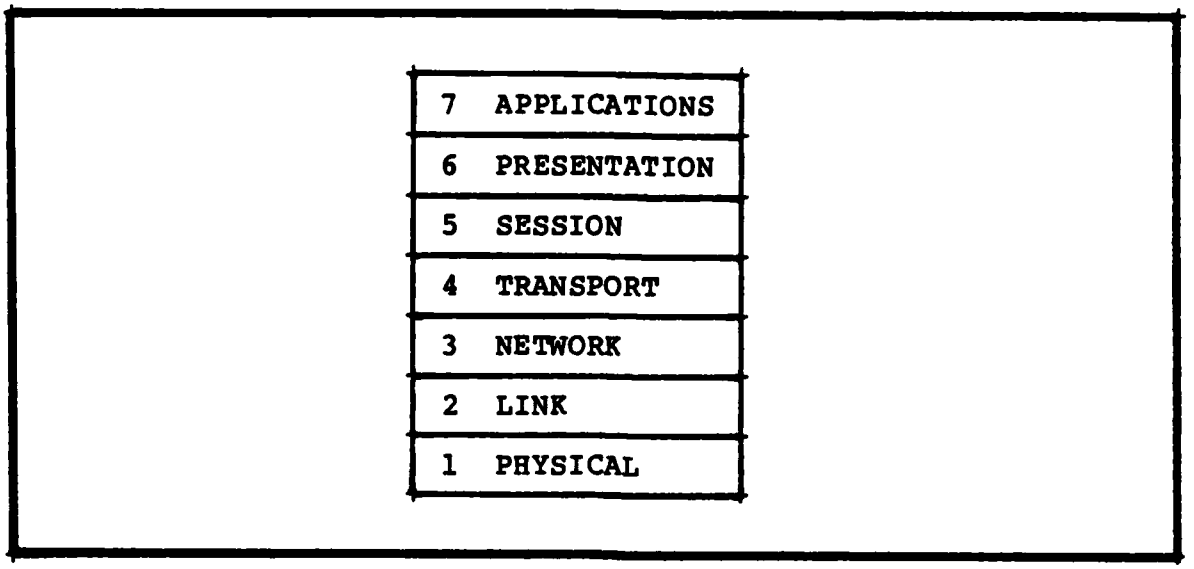


Figure 2-1. ISO Open Systems Interconnection Reference Model (Ref 11:8)

foreseen. Flexibility, as it applies to the I/O interface, specifically addresses the ease with which new devices or new configurations can be adapted to the structure of the interface. This is especially true in relation to the installation of different processors or peripheral devices into the system. The protocol must be completely independent of the type, number, or organization of the processors on which it operates. However, it must be capable of working in all future processor configurations.

Conceptual Standards. Communication among interconnected computer systems can be viewed as seven levels. The International Standards Organization (ISO) Open Systems Interconnection Reference Model identifies these layers of communication protocol (see Figure 2-1). The

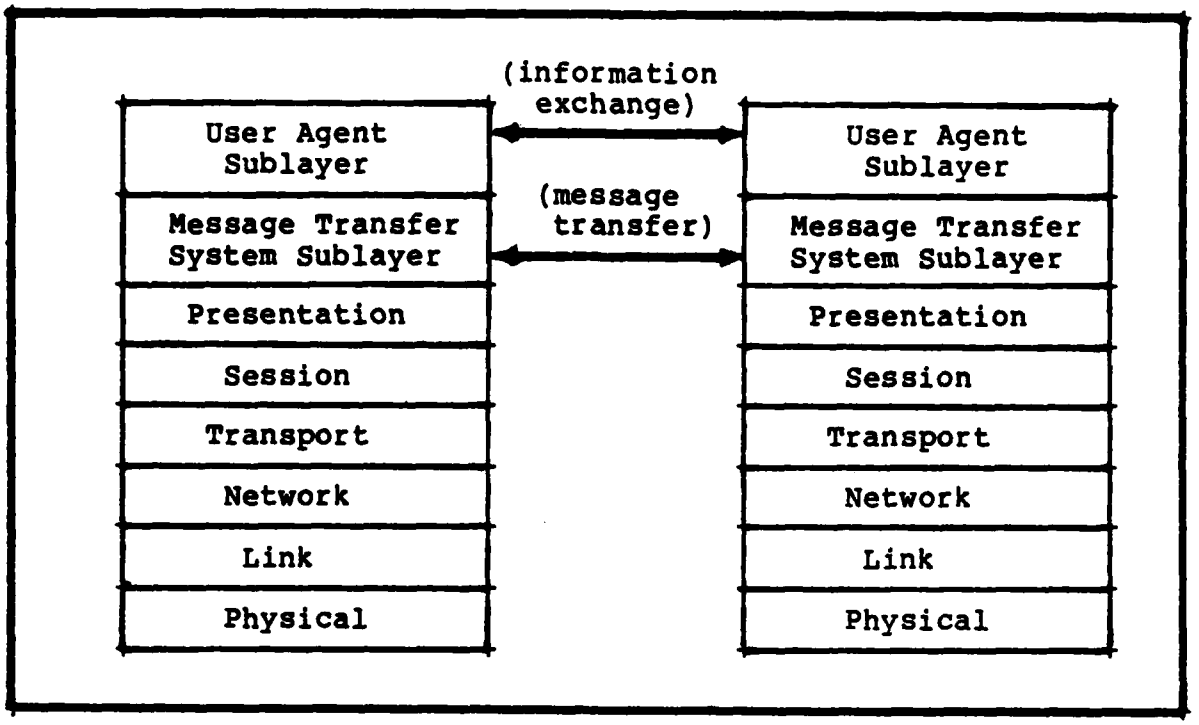


Figure 2-2. Message Transfer Protocols in the ISO Reference Model

lowest levels (Network, Link, and Physical), are generally the concern of local area networking. There are existing standards for system interconnection at these levels (e.g., EIA's RS-449, CCITT's X.25). Higher levels are not as well defined, however, some work is being done in this area by the Institute for Computer Sciences and Technology of the National Bureau of Standards (Ref 11:8-9). This work includes several reports on the features and specification of a computer based message system (CBMS) (Ref 11,10,9,8).

A CBMS is an application level system which allows its users to prepare, manage, send, and receive messages (Ref 8:2-5, 10:2-4). Message transfer protocols are used

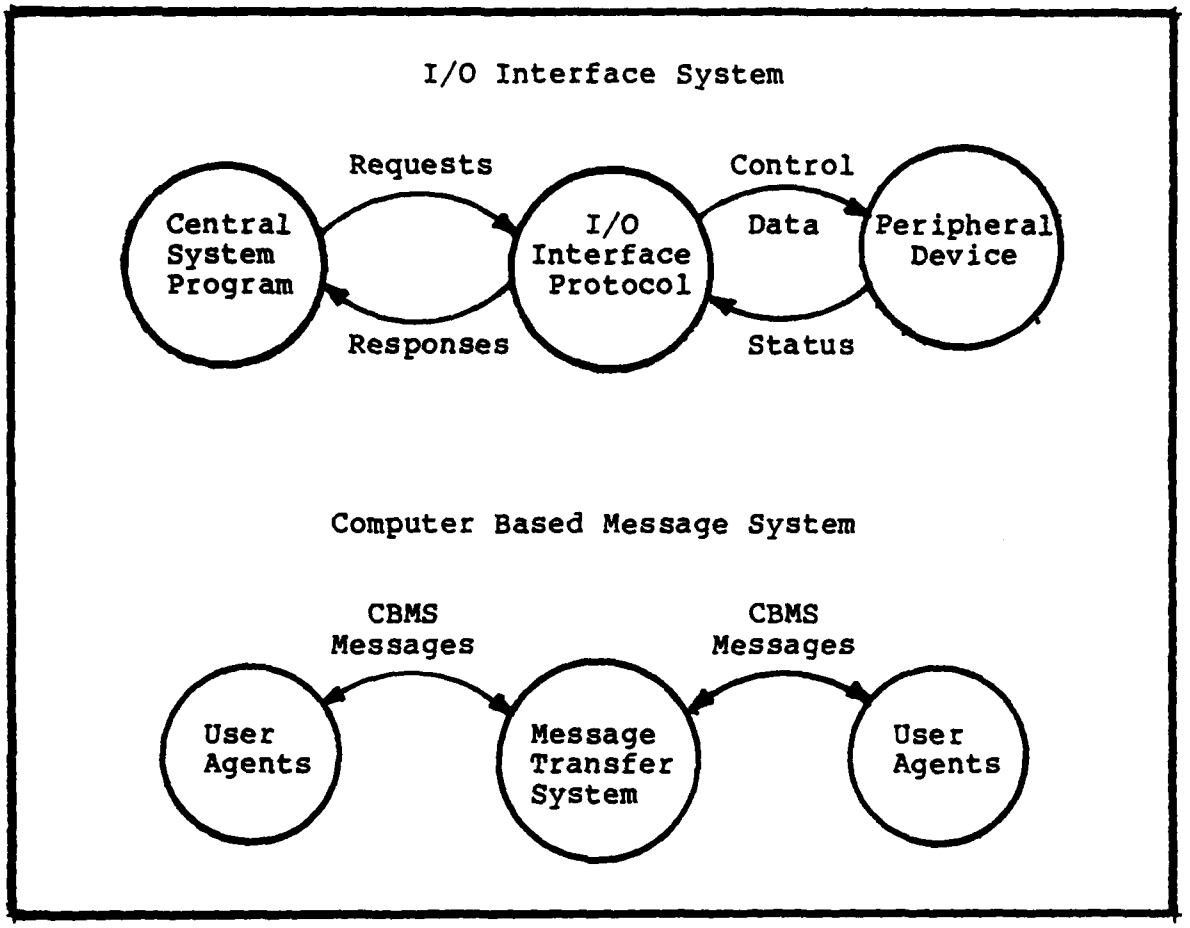


Figure 2-3. CBMS Within the I/O Interface Protocol

when a message is moved from one CBMS to another. The model for a CBMS divides the components of the system into two classes, User Agents (UAs) and the Message Transfer System (MTS). The UAs provide the message creation, display and management services to the CBMS users. The MTS provides the functions necessary to transfer messages between different UAs. Figure 2-2 shows the CBMS classes with respect to the ISO reference model. (Ref 8:1-6). The use of the CBMS within the I/O Interface is shown in Figure 2-3.

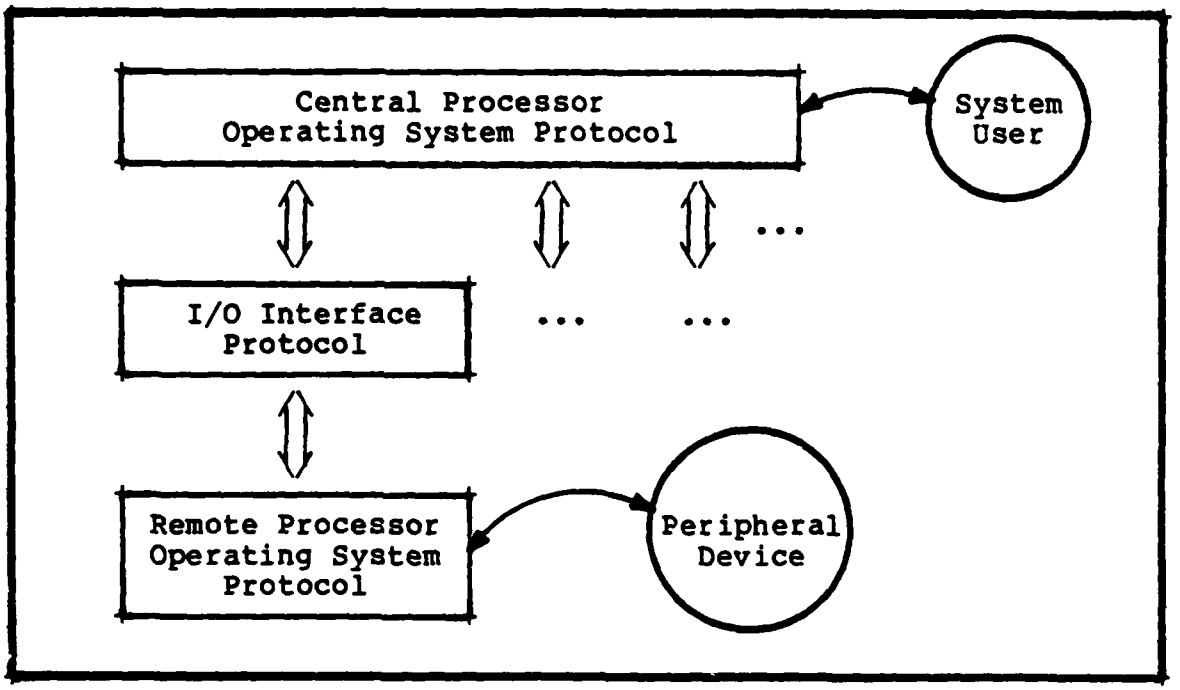


Figure 2-4. I/O Interface Protocol Relationship to the Other Protocols

Depending upon the size, complexity, and specific purpose of the network, the three top levels of the ISO reference model (applications, presentation, and session) may be blurred as to their specific definitions. However, the CBMS protocols clearly belong above the transport layer and depend upon the services of that layer to function. A strict presentation layer, however, is not required (Ref 8:6-7).

Functional Requirements for the I/O Interface Protocol

The functional requirements for the I/O Interface protocol are presented from several viewpoints for clarity. These views include the protocol in relation to the entire

system, a model of the protocol operation and the functional description of the protocol.

Relation to the System. As shown in Figure 2-4, the I/O Interface protocol communicates with the operating system protocols of the central processor and remote processor. The remote system communicates directly with the peripheral devices.

Model of Operation. The model of operation for performing an I/O function through the remote processor is illustrated by the following (see Figure 2-5):

1. The user's application program prepares the data and calls the I/O Interface to create and transfer the message to the appropriate remote system which controls the desired peripheral device. The data, destination address, and other parameters are passed as arguments of the call.
2. The UA module of the I/O Interface prepares the message format. The MTS module then determines the inter-processor port for the given system address, in this case, assume it is the address of a printer.
3. The MTS then sends the message to the I/O port connected with the remote system controlling the requested printer. Operating system functions are used for this transfer.

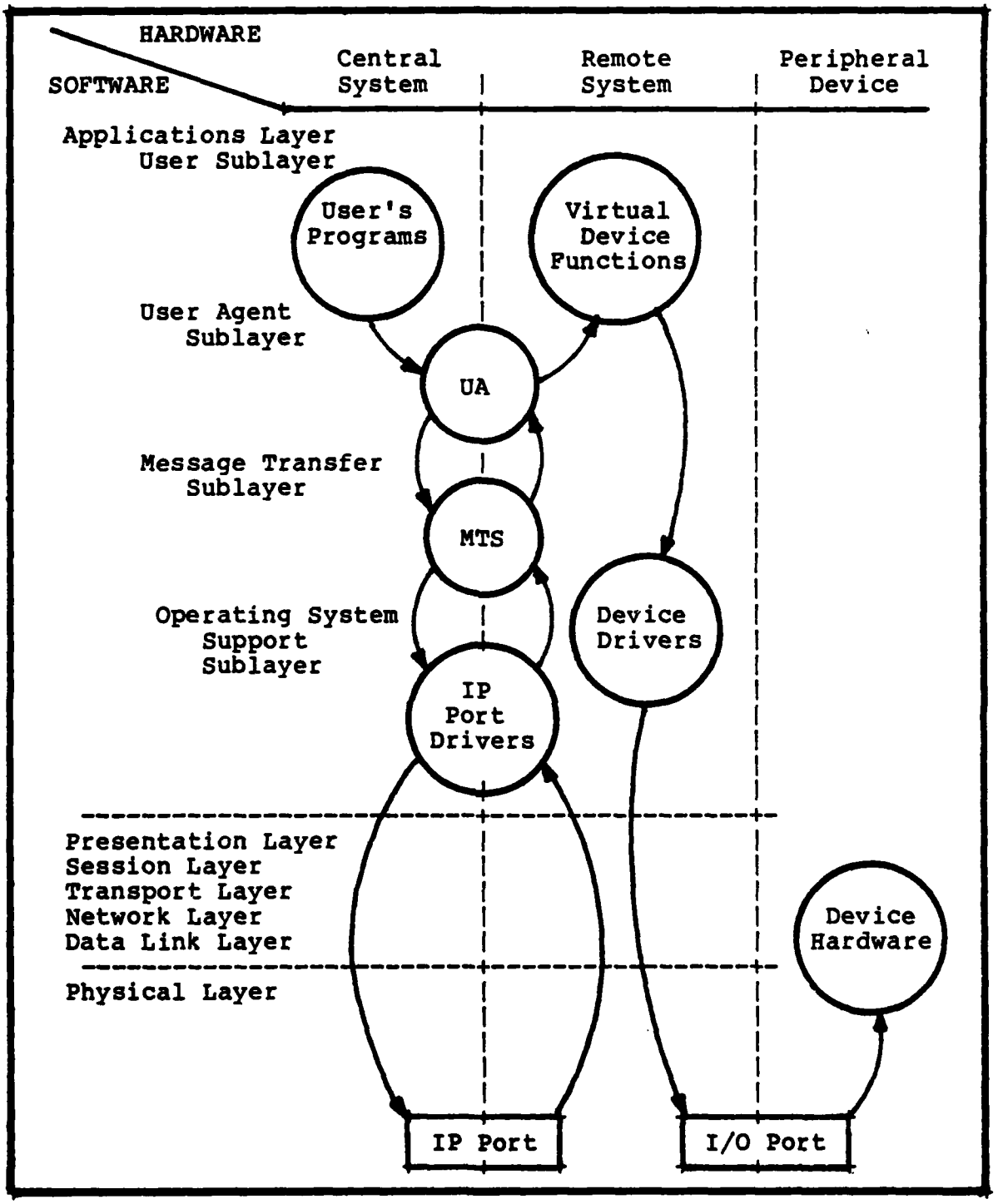


Figure 2-5. Model of I/O Interface Protocol Organization Within ISO Reference Model

4. The remote system interface reads the control parameters of the message (MTS level examining the message envelope) and sends the message to the UA servicing the process which handles the printer operation.

5. Upon completion of the task by the peripheral device, the status of the operation is returned as a reply message created by the UA. This reply message is then sent back to the originating process.

6. The User Application Program then receives the reply and determines if further action is necessary based on the status of the operation as indicated in the message. The status information is mapped into the appropriate error message or status code for the applications program or operating system.

Figure 2-5 shows this process as a data flow among system modules. The physical system boundaries are indicated by vertical dotted lines in the figure. The ISO Reference Model layers are shown by horizontal divisions. The interface to another network would be done at the Transport Layer of the ISO model. For example, the Digital Engineering Laboratory's Network (DELNET) provides the services of the lower three layers of the ISO model. Interconnection to this subnetwork implementation would require a Transport Layer protocol be provided within one processor system and a I/O Interface system user entity

created to communicate with the protocol functions.

Functional Description. The functional requirements for the I/O Interface are presented as a set of features, or mechanisms. Structured Analysis and Design Technique diagrams (SADTs) and Data Flow Diagrams (DFDs) will not be used because these requirements are not necessarily functional in nature. The I/O Interface functional requirements are characteristic features of the interface system and may be defined within one physical system or several, and within one software module or several. The next chapter will use DFDs to describe the design of the I/O interface for the Intel 432 Micromainframe computer system.

The overall purpose of the I/O Interface protocol is to move data from a central system to a peripheral device. This is done using a CBMS between the device user process and the device driver process. The interface function modules reside in the central and remote processors of the system. The data is routed from one module to another until the device driver is reached. The reply, giving the status of the operation, is then transmitted back along the same path. Thus, one important mechanism of the I/O Interface protocol is the I/O device addressing.

To have effective communication between a sender and a receiver, it is necessary to have a well-defined message system. The structure of the message must be clearly understood by both the originator and the recipient of the

message. The sender must be able to create a structured message so that the receiver can extract the meaning. The formatting of messages is a necessary mechanism of the I/O Interface protocol.

The receiver of a message may be given control information indicating the purpose of the data in the message. This can extend the usefulness of the of the message system by allowing multiple formats for data transfer. When the message arrives at the UA, the receiving process may use the control information to determine the format of the remainder of the message and even the function to be performed by the user process. Thus, a control mechanism is important in defining the functions available to users of the interface.

Likewise, the success or failure of the operation is important to the process which originated it. The receiving process must have a method of communicating the status of the operation to the requesting process. The protocol provides a mechanism for reply, defining the responses which may be made by a device serving the interface.

The following paragraphs discuss these mechanisms required for the interface protocol; address, format, control, and reply.

Address Mechanism. A distinction is made in the address mechanism between names, addresses, and routes (Ref

9:9). A name is an identifier of a resource. An address is the specific location of that resource. A route is a path between two resources. The I/O Interface implements a mapping from names to addresses.

The mapping function can be performed by the originating UAs or by their associated MTS agents. The information relating the name to its address is stored in a CBMS directory. This directory is logically a single, centralized database. However, it may be distributed and partially redundant. That is, the process of looking up a name may be done in a distributed manner. In that way, each MTS only determines enough of the address to know the next destination in the route (i.e., the next MTS or the destination UA).

Format Mechanism. The message format specification describes the form and meaning of messages as they are sent by one CBMS and received by another. Messages are generally composed of "fields" (segments of a message), containing different types of information. These types include names of the originator and receiver, subject data, security classification, references to previous messages, as well as the text of the message. Standard syntax for messages provides a means for the contents of messages from one CBMS to be retrieved by another CBMS. Standard semantics for the different types of information allows consistent interpretation of the message by the receiver. A

definition of the message syntax and semantics is required for the I/O Interface.

Control Mechanism. The control mechanism is the highest definition of function in the interface. It may be used to indicate the active fields within the message, the particular function requested of the receiver, or the type of handling required. This control mechanism may be implemented as a field of the message (UA level) or the message envelope (MTS level).

If the control is placed on the envelope, then the information would not generally be available to the UA receiving the message. The MTS would place the message at a particular UA depending upon the control field contents. Where messages were extremely complex, such a system would allow the UAs to be simplified, each handling only a single message format or function.

Placing the control field in the message would allow the control data to be used as a function control for the user, thus, the message might only consist of the control field. For example, a request for a "form feed" on a printer device requires no additional data in the request message. However, each UA must able to process messages of all types. While many may not apply to a particular user, all control values must be handled properly, generating the appropriate response. This requires greater attention to generating responses for improper requests.

The control mechanism must allow for expansion. It must be defined to allow new functions or message types to be incorporated within the existing organization. New formats or functions may be required by future users of the interface.

Reply Mechanism. The reply mechanism is the method by which a receiver reports the status of the requested function to the sending process. A distinction must be made between the response message, which may be generated by a receiver as a result of a particular function, and the status reply message. A "read" request, for example, will result in a message containing the input data being sent to the requester. This is only a reply if the message also contains status information on the condition of the read operation.

The status information may be provided to the user by the UAs or their associated MTS agents. The status information should be encoded to reduce the length of the message necessary for the reply. The meaning of the code should be available to all users from a common "error message directory". This directory is logically a single database, as with the addressing information, and may be distributed. Each system may then have its own directory of error messages which apply to those users or devices with which the system interfaces. Generally, this will require duplication of most of the directory from one system, to the

next, to allow freedom of communication among the systems.

As with the control mechanism, flexibility in the reply mechanism is necessary. New uses may require additional status information to indicate the condition of their functions. The reply mechanism must be able to expand to meet future needs.

Functionally, the I/O Interface protocol provides for addressing, formatting, control, and reply, among the user processes or devices using the interface. Implementation of the protocol is restricted by the limitations of the current 432 system at AFIT.

Functional Standards. Standards for the functions of the I/O Interface are those which apply to Computer Based Message Systems (CBMSs) in general. The National Bureau of Standards (NBS) is preparing standards for CBMS systems at this time. Their work includes a proposed standard for message formats (Ref 12) and a report on naming and addressing in CBMS systems (Ref 9). In addition, standards have been adapted for network addressing using the AFIT Universal Network Interface Device (UNID) developed here at AFIT (Ref 32). This latter work is important because it is proposed that most of the small computers at AFIT will be connected to a subnetwork of UNIDs in the near future. In general, these proposed standards for interfacing message systems place functional requirements on the I/O Interface design in two areas; user address, and message format.

User Address. An address consists of a series of attributes defining a location relative to the MTS layer. A general discussion of the attributes which can be used in addressing and their representation is presented in the report "Naming and Addressing in Computer Based Message Systems," written for the National Bureau of Standards (NBS), Institute for Computer Sciences and Technology (Ref 9:30-33). The general structure of the addressing mechanism presented in this document suggests an address with four attributes to indicate the specific address of a UA relative to the system; country, network, host, and user-id. In addition, to have compatibility with the DELNET subnetwork system proposed for AFIT computer systems, the addresses structure must match that expected by the Universal Network Interface Devices (UNIDs).

The addressing system used in the UNIDs is compatible with the proposed NBS system using country, network, host, and user-id attributes (Ref 9:33-34, 32:3-5/3-9). In the UNID system, the user-id is called the "port-code" as a designation of a particular I/O port address on a microprocessor. Details of this addressing scheme are presented in Appendix E.

Message format. A general description of the syntax and semantics for CBMS messages is presented in the Proposed Federal Information Processing Standard "Specification for Message Format for Computer Based Message

TABLE 2-I

I/O Interface Message Fields

Originator Fields	
FROM	Required
REPLY-TO	Basic
Date Field	
POSTED-DATE	Required
Recipient Field	
TO	Required
Message Content Fields	
SUBJECT	Basic
TEXT	Basic

Systems" (Ref 12:20-56). This section presents the requirements for the I/O Interface system message format.

The message fields required for the I/O Interface are listed in Table 2-I. These fields include those which are required by the CBMS format specification and those optional fields which are necessary or basic to the functioning of the I/O Interface. Many other fields are possible within the CBMS format as defined by the NBS proposed standard (Ref 12:15-34) and may be required by a CBMS with a larger scope of action that of the I/O interface system developed here.

The CBMS message consists of two types of components, fields and messages. Fields correspond to the semantic components described above. A message is simply another message. The type of field determines its meaning and organization of its contents. The fields are composed of

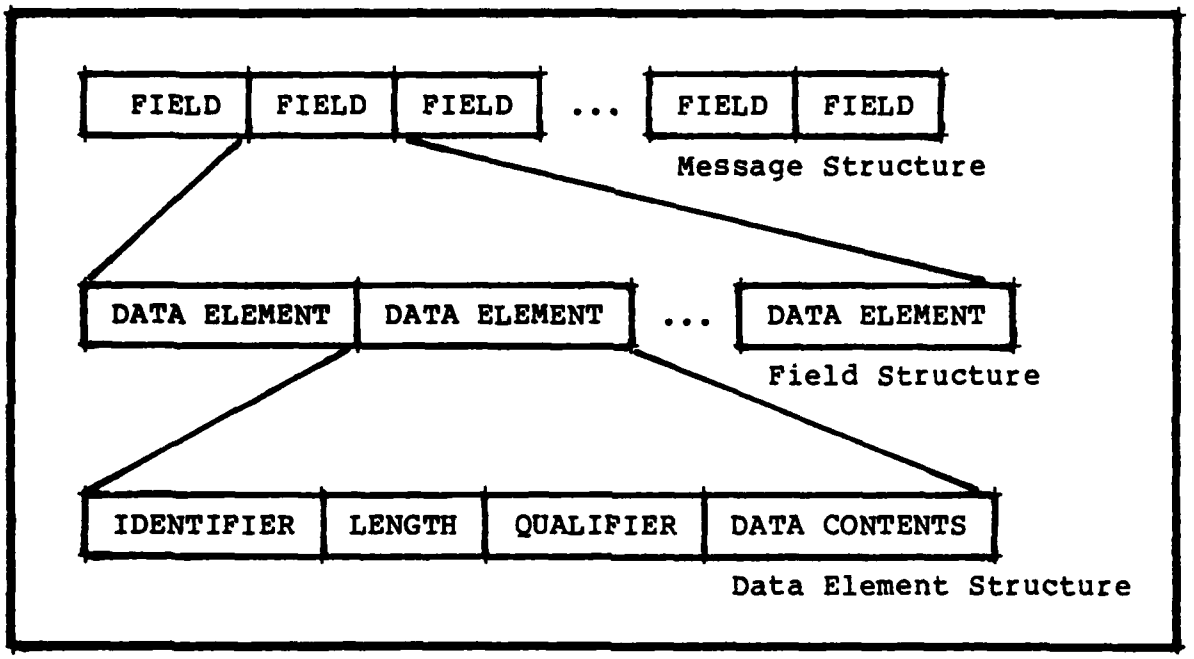


Figure 2-6. CBMS Message Structure

data elements which have, at most, four components; Identifier, Length, Qualifier, and Data Contents. Figure 2-6 shows the organization of the components of a data element.

The standard syntax also includes an optional "Property List" component, which may contain a printing name (label) for the data element or one, or more, comments. This information is not necessary for the operation of the I/O Interface and, therefore, use of the "Property List" element is not required for the implementation.

The Identifier is always the first byte of the data element. It is one octet (8 bits) in length and indicates the type of the data element. The next component is the

Length. This is an unsigned integer giving the number of octets that appear following it in the data element. The Qualifier component supplies additional information to identify the element and, finally, the Data Contents contain the actual data of the element. Details of the structure are presented in Appendix D.

Implementation Constraints

The constraints on the implementation of I/O interface on the Intel 432 Micromainframe computer system are primarily due to its immature state of development of the system. The 432 system design is continually being updated as improvements are made to the hardware and software. To allow implementation design work to be completed, the configuration of the 432 system must be frozen at some point. For purposes of this thesis effort, the fixed environment is the hardware and software compatible with the "Release 2.1" hardware components (see Appendix F). The following sections will discuss the limitations of the hardware and software in this environment.

Hardware. There are two hardware systems to consider with the Intel 432 system. These are the configuration of the 432/670 Micromainframe computer system (432) and the components of the Cross Development System (CDS). The limitations of each place constraints on the development of the I/O Interface implementation.

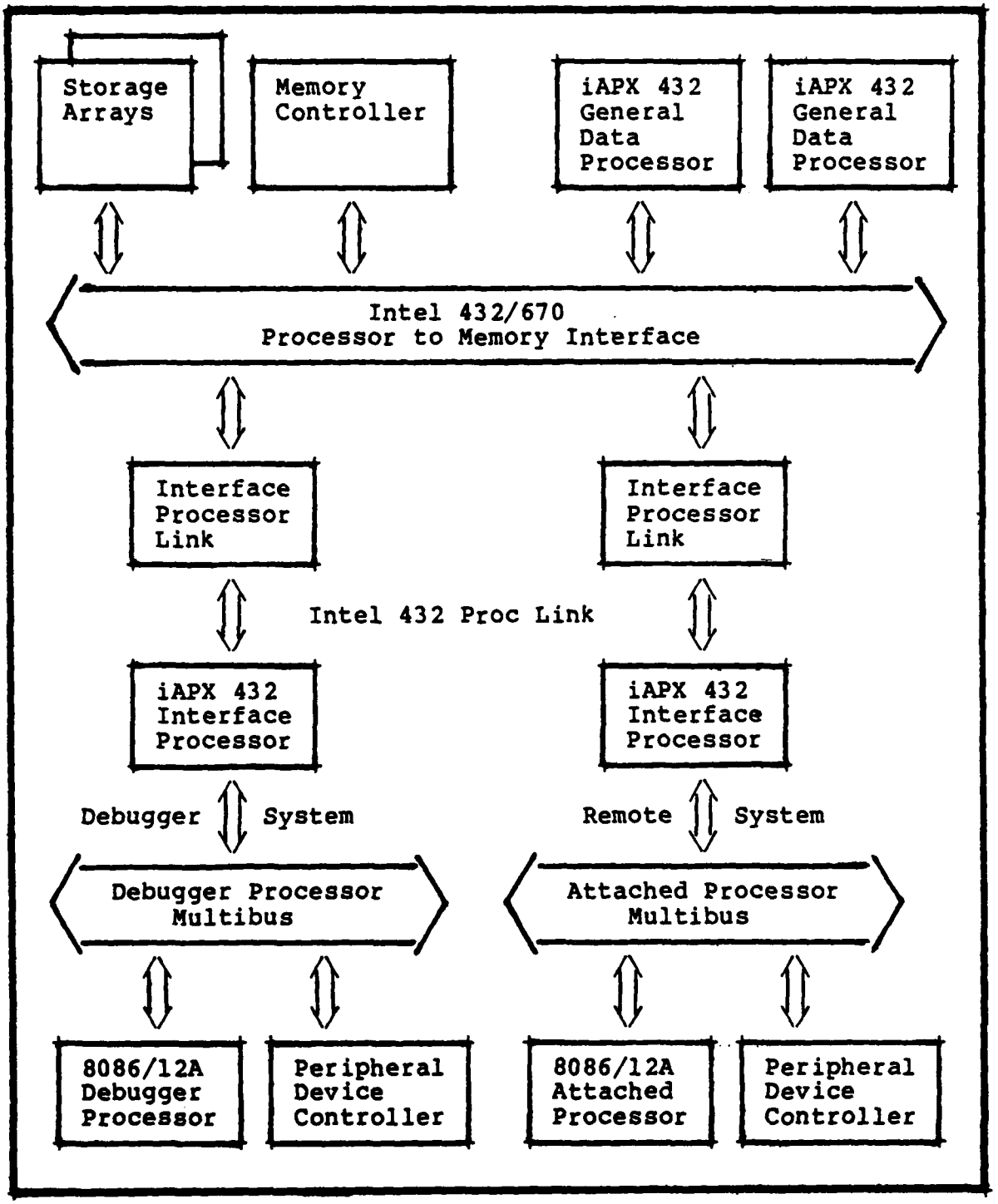


Figure 2-7. Intel 432/670 Micromainframe Hardware Configuration

432 Micromainframe. The 432 is complex multiprocessor system. As explained in Appendix A, the system contains multiple General Data Processors (GDPs) and communicates with the outside world through Interface Processor (IP) communication links with Attached Processor subsystems (APs), which may be any 8 or 16 bit microprocessor systems using MULTIBUS, IEEE standard 796, interface (Ref 24:1-3).

These AP subsystems perform all the input and output functions for the 432 computer system. One of the I/O subsystems is the debugger, or Debug Workstation (Ref 22:1-2/1-4). This Debug Workstation processor system is not usable for any other purpose with the 432 while it acts as the system debugger. Therefore, implementation of an I/O Interface to a functioning device requires a second AP subsystem with its own IP system (see Figure 2-7). Currently, only one IP board has been modified to be compatible with the Release 2.1 configuration (Ref 35:A-2/A-4). Until the second IP board is upgraded, the system is constrained to only a single AP system which must be the debugger. This means that it is impossible to fully test the interface. There is no way to communicate between the 432 processor and an AP system running the I/O Interface software. Limited testing of the software is still possible without system interconnection.

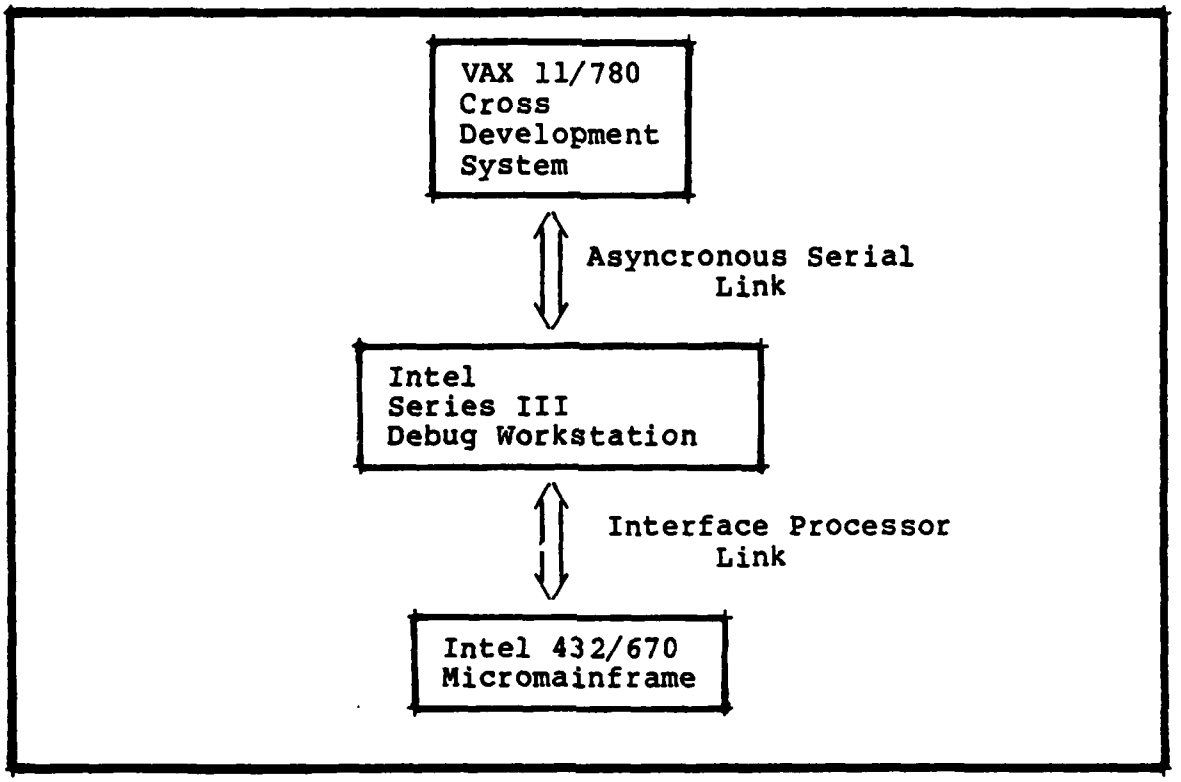


Figure 2-8. The Intel 432 Cross Development System

432 Cross Development System. The cross development system for the 432 involves three distinct computer systems. First, the compiler and linker for the Intel 432 execute on a VAX 11/780 computer system. Second, executable code must be transmitted to a debugger system (an 8086-based Intel Series III Microcomputer Development System) for loading into the third system, the 432/670 Micromainframe (Ref 22:1-3/1-6). Figure 2-8 shows this organization. Limitations in the operational condition of any of these systems constrains the software development process for the 432.

Specifically, the Intel Series III, Debug Workstation, is required to have at least one hard disk system for full operation. The available systems at AFIT currently do not have a hard disk. This limits the maximum size of the executable code file to the space available on an ISIS double density floppy disk (3895 blocks of 128 bytes each). This limits the development of 432 processor software. The iMAX operating system run-time environment is nearly 3000 blocks alone. The AP environment is not a problem because the iRMX 88 run-time system is configurable and need only contain the modules necessary to support system calls that are made. Until a hard disk system becomes available, it will be necessary to minimize the size of the modules which must be linked together for execution on the 432. The 432 processor software must be designed to be tested in this limited environment.

These hardware restrictions will eventually be overcome and, thus, do not permanently negate the value of this design effort. The hard-disk system has been promised by Intel Corporation as a part of the 432 system package which will provide an environment for development of productive software systems. The Interface Processor board is being upgraded by Intel technicians and will be returned as soon as possible. As larger projects require more hardware, the 432 configuration may be expanded to include several additional IPs providing interface with a large variety of

AP systems.

Software. Software development for the 432 system is highly constrained. There is only one programming language available for the 432 processor (Ada). Selection of the language used with the AP (8086-based system) is limited by the 432s operating system (iMAX), which is provided as a set of PL/M-86 source code programs. In addition, compatibility must be maintained among the utility programs (compiler, linker, debugger, and operating system) of the 432 Cross Development System (see Appendix F). These facts leave little room for discussion of the optimum programming language for this work.

432 Software. A cross compiler for the Ada programming language is the only compiler available for the Intel 432 computer. The current version of the compiler does not implement the full Ada language as defined by MIL-STD-1815A (Ref 1). The restrictions on the Ada language for the current compiler are listed in the "Intel 432 VAX Host Users Guide" (Ref 21:A-1/A-4) and summarized in Appendix G.

Extensions to the Ada language which permit more efficient use of the features of the iAPX 432's instruction set have been included in the compiler implementation. The extensions to Ada are detailed in the Intel Reference Manual for the Intel 432 Extensions to Ada (Ref 27). Generally, it would be advisable not to use extensions to an established

language for the software development to avoid problems with portability and the non-standard aspects of these extensions. However, since the compiler is not fully implemented and the system structure is unusual, use of the language extensions could allow more efficient implementation of the software. The software is clearly annotated where these extended features have been used. Future implementations and modifications to the software will make use of compiler features then available and use proper Ada language constructs to perform these functions.

Attached Processor Software. The languages available for the 8086 Attached Processor system are also limited. While several compiler systems are available which create 8086 executable code, the program modules which provide for control of the Interface Processor are written in PL/M-86. Also, the I/O functions of the iMAX operating system are implemented on the AP using the executive features of iRMX 88 Real-Time Multiprogramming Executive (Ref 25:IOI-1). The software provided by Intel for linking with these operating system functions is written in PLM-86. A compiler system for PLM-86 language is available on the Intel Series III Development System and programs may be written which are compatible with modules written in assembly language (ASM-86). A linker (LINK-86) is provided to combine assembled or compiled program modules and library functions into executable files.

Two program languages are used in this thesis effort; Ada, for the 432 processor, and PL/M-86 for the 8086-based AP. The top-down approach was used in the development and testing of the Ada software to permit the greatest possible progress towards a fully functioning interface system (i.e., largest possible executable module that can be transferred to the Debuggers disk system). Where a choice existed, Ada language facilities are used, in favor of iMAX operating system structures, to provide compatibility with future software developments, using improved Ada compilers. Also, the structuring of the PL/M-86 software, for the AP, is similar to the Ada language software (object-oriented) to maintain system continuity.

Summary

This chapter has defined the requirements for the I/O Interface protocol and the implementation of the protocol on the Intel 432 Micromainframe computer system from the conceptual and functional viewpoints. These requirements are summarized in Table 2-II. The conceptual requirements focused on the virtual operation and system flexibility. The functional requirements were defined for the I/O Interface protocol including mechanisms for addressing, formatting, device control, and reply. Finally, the constraints on implementing the protocol using the Intel 432 Micromainframe were discussed. The next chapter presents the design for the I/O Interface protocol.

TABLE 2-II

Summary of I/O Interface Requirements

- Concepts
  - Virtual Device Operation
  - System Flexibility
- Structural Models
  - ISO Open Systems Interconnection Reference Model
  - NBS Computer Based Message System Model
- Functional Mechanisms
  - Addressing
  - Formatting
  - Device Control
  - Device Reply
- Implementation Constraints
  - Lack of Interface Processor for Attached Processor System
  - Lack of Hard Disk for Program storage in Debugger System
  - Incomplete Ada Language Compiler

III. SYSTEM DESIGN/IMPLEMENTATION

Introduction

The first two chapters have presented the objectives and requirements for the I/O Interface on the Intel 432 Micromainframe computer system. Chapter I made reference to two tools for a successful design effort with the 432 system. These are the use of top-down structured implementation and the object-oriented design methodology.

This chapter deals with the actual design of the interface functions as implemented in 432 software. The design is approached from the user's viewpoint, dealing initially with the highest level of functionality which must be provided to the using process or device. These functions are referred to as the "services" provided by that level. The design proceeds to the lowest level where iMAX operating system services are used to perform the required services.

Before discussing the top-down development of the I/O Interface, it is important to look, again, at the concepts and functions required for the interface. The next section describes the general features of the system design, which implement the conceptual requirements. The following sections, then, present the implementation of the functional mechanisms of the I/O Interface and finally, the design of the system within the structure of the ISO Reference Model.

General Design Features

The general design features of the I/O Interface include the structure of the system processes and the conceptual requirements presented in Chapter II; virtual devices, and system flexibility. The method of implementation for each of these features is presented in the following paragraphs:

System Processes. The processes implemented on the 432 processor are created as "static" tasks under the iMAX operating system. This means that the task cannot be created or destroyed during system operation (refer to Appendix C for more information on tasks). This was done because the Ada language compiler (Version 2.0) does not support the tasking facilities of the language defined by the DOD manual (Ref 21:A-1/A-4). The use of the tasking facilities provided by iMAX (Ref 25:BPM-1/BPM-19) would create a more powerful environment, but, restrict the system to using the iMAX process management system with all future modifications. This is because, in general, it is advised not to use the tasking facilities of Ada and iMAX together due to the differences in process data structures (Ref 25:BPM-1).

For the I/O Interface system, this means that users may not be added or removed while the system is executing. In a multiuser environment, it would be necessary to start a shell process for each user. This was unnecessary for

TABLE 3-I

I/O Interface Functions and Reply Codes

I/O Interface Functions		I/O Interface Reply Codes	
0	Open	0	Ok
1	Close	1	Invalid Command
2	Read	2	End of File
3	Write	3	Bad Data
4	Page	4	Device Error
5	Title	5	Device Closed
6	Delete	6	Device Offline
7	Rename	7	Device Busy
8	Reset	8-255	Reserved for Future Use
9	Get Config		
10	Set Config		
11	Test		
12-255	Reserved for Future Use		

simply testing the I/O Interface. The result is that the system is strictly single-user. However, on the AP system there is dynamic creation of the tasks for each device. This would allow new devices to be added to the system during execution. This function may be implemented by future designers.

Virtual Devices. The implementation of the I/O Interface protocol presents a set of virtual devices to the user. These devices are designated by a name and address which uniquely specify the location of that device within the complete system. The user's view of the interface is a set of procedures which perform I/O functions. The user provides the name of the device as an argument in the

procedure call. Thus, the set of procedures forms an abstraction, or virtual mechanism, of the system I/O devices.

Each of these procedures also requires a reference to an integer variable which may be modified by the routine. This variable is set to a "reply code" which is a status indication from the device. Table 3-I shows the available functions and the reply codes which may be returned. The virtual device created by the I/O Interface will, therefore, respond to a known set of requests with a status indication from a known set of values.

This virtual device representation is implemented in the User Agent level of the CBMS structure presented in Chapter II. This is the point at which the user applications programs would have access to the CBMS communications system. This organization makes the CBMS system a tool of the I/O Interface and allows the user access to only the functions of the I/O Interface. The use of the CBMS is effectively hidden from the user's view. The details of the implementation are presented later in this chapter.

System Flexibility. Flexibility of the system design is provided on two levels. First, flexibility is provided in the software design by using the object-oriented design methodology described in Appendix B. This modular design approach protects information about the implementation of data structures within software modules. This allows the

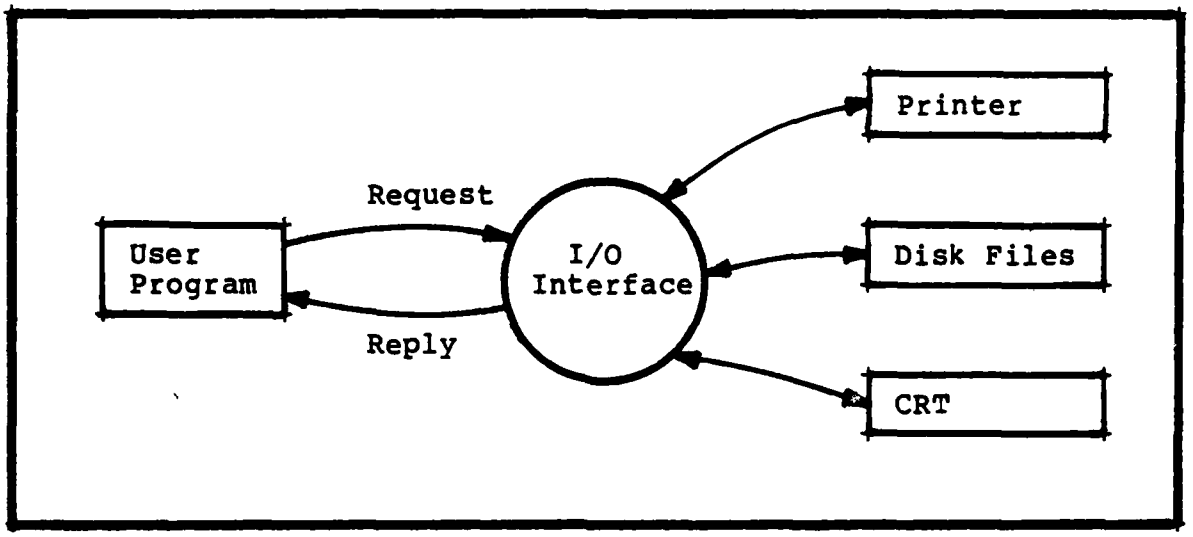


Figure 3-1. I/O Interface Context Diagram

system designer to modify the implementation with minimum effects on other parts of the software system (Ref 17:6-23). The second level of flexibility is in the implementation of the control and reply mechanisms for the I/O Interface. Essentially, the user's view of the interface is the representation of several copies of a single type of device which accepts a known set of commands and responds with one of a known set of replies. Any new device would be used in the same manner, and thus, be made immediately available to all users through the same interface. Figure 3-1 shows this in a context diagram. The I/O Interface appears to the user as a single set of functions which allow access to all the devices in the system.

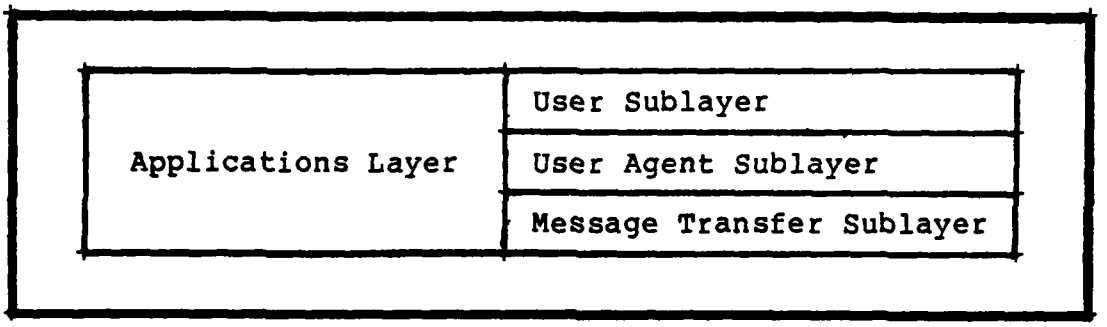


Figure 3-2. I/O Interface Sublayers within the ISO Applications Layer

Functional Mechanism Implementation

Chapter II presented four mechanisms required for the I/O Interface; addressing, formatting, device control and device reply. This section describes the location of these mechanisms within the organization of the I/O Interface and the ISO Open System Interconnection Reference Model.

The I/O Interface is implemented within the Applications layer of the ISO Reference Model. The requirements for the interface have divided this ISO layer into sublayers. These sublayers are shown in Figure 3-2.

Address Mechanism. The addressing mechanism is implemented in the Message Transfer Sublayer. The mapping of device names to device addresses, referred to as "CBMS addresses", must be used by the MTS procedures to determine the correct route for the destination requested. The CBMS address may also be included as a part of the device name itself according to the NBS report describing naming and addressing in CBMS systems (Ref 9:23-36). However, the I/O

Interface only uses the CBMS address to determine the appropriate route for the message.

The Message Transfer Sublayer is given the destination of the message in the form of a device name which it maps to a CBMS address. But the CBMS address is only a binary code for the precise location of the device and the MTS process requires the local system address of a communications port where the message should be sent. So there is a second mapping function which takes the address of the device and determines the communications port to use for that device relative to the current location. Both these mapping functions are presented in detail in the section describing the Message Transfer Sublayer.

Format Mechanism. The procedures which handle the format of the CBMS messages are all contained in one software package in the Message Transfer Sublayer. The message structure is not available to direct access by any other procedures. This effectively hides the CBMS structure from the rest of the system. Message information may only be accessed by the procedures provided in the package and any changes to the structure of the messages does not affect any software outside the package. This is an example of the object-hiding feature of object-oriented design (see Appendix B).

Implementation of the format mechanism is presented with the Message Transfer Sublayer later in this chapter.

The details of the message structure, the CBMS message format, are presented in Appendix D.

Device Control Mechanism. The device control mechanism is half of the implementation of the virtual device feature described above. The device control procedures available have been listed in Table 3-I above. These procedures are found in the implementation of the User Agent Sublayer of the I/O Interface.

Device Reply Mechanism. The device reply mechanism completes the virtual device implementation of the I/O Interface and is also located in the User Agent Sublayer. The reply codes returned by the control procedures are also listed in Table 3-I.

The following sections discuss the design of each sublayer and the software packages which implement these mechanisms described above.

User Sublayer

The User Sublayer is the highest level of software in the I/O Interface implementation. In fact, this layer is not actually a part of the interface proper. The User Sublayer contains the processes and devices which use the interface for communication. This level is implemented as a part of this thesis effort to demonstrate the functioning of the I/O Interface and show, by example, how processes and

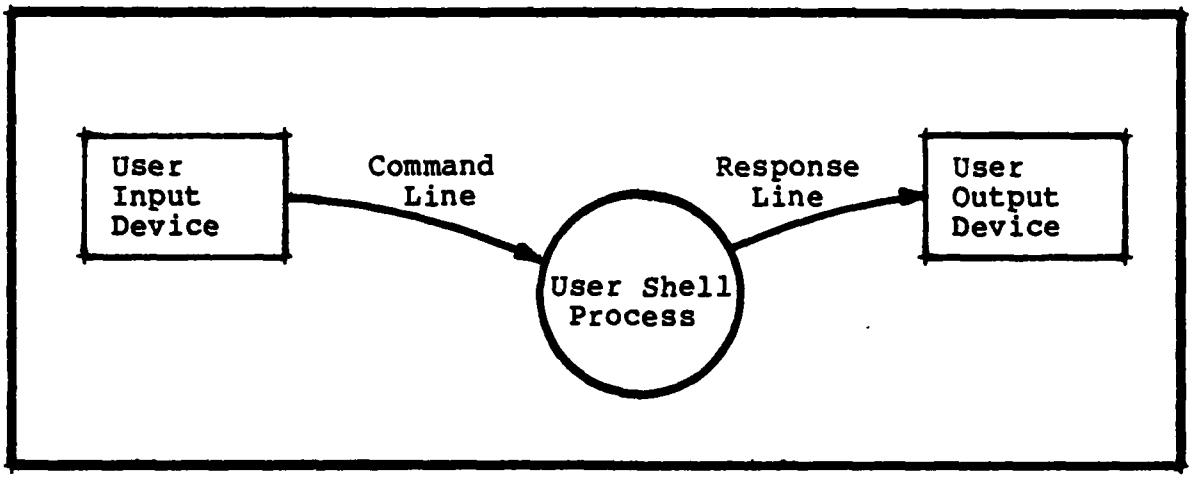


Figure 3-3. User Shell Context Diagram

devices interact with the interface. On the 432 processor, within the User Sublayer, a minimal "shell" process is implemented which accepts user commands and communicates with other system devices using the I/O Interface. On the AP system, the User Sublayer contains the device abstractions with which the User Shell can communicate.

User Shell. The User Shell is a process which executes on the 432 processor. There are two significant data structures in the implementation of the shell. First, the syntax of the command line is defined and managed by the procedures of the User Interface package. Second, the system commands available to the user are defined and controlled by the procedures of the System Commands package. The third software package of the User Shell is simply called Shell and contains the procedure Main which controls the sequencing of action in the Shell. The Shell package is

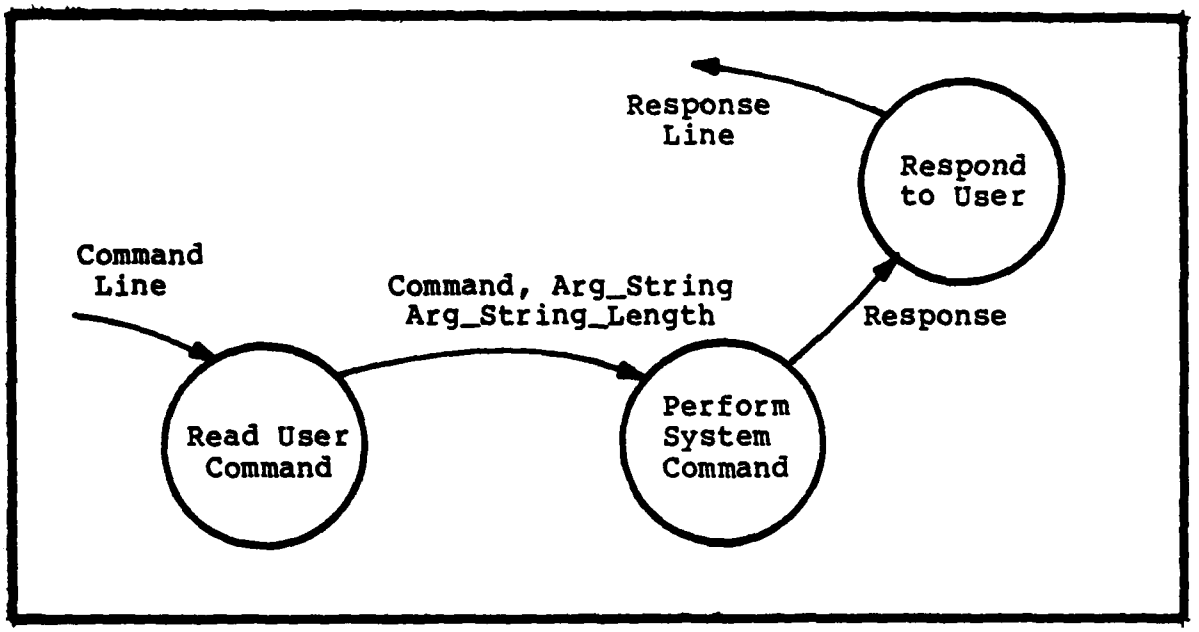


Figure 3-4. User Shell Main Program Data Flow

presented first, followed by the User Interface and System Commands packages.

Shell. The shell is implemented as a simple command processing loop. The Shell package of the User Shell contains only one procedure, Main, which is the primary control loop calling functions provided by other packages to perform the requested commands.

Figure 3-4 shows the data flow within the Main program loop. The Read_User_Command and Respond_to_User procedures are part of the User Interface package. The Perform_System_Command procedure in the System Commands package.

TABLE 3-II

Procedures of the User Interface Package

1.1 Read_User_Command 1.2 Get_Argument 1.3 Read_from_User 1.4 Write_to_User 1.5 Respond_to_User

User Interface. All interface with the system operator or user is done through the User Interface package. The package contains several procedures which create an abstraction of the user (see Table 3-II); Read_User_Command, Get_Argument, Read_from_User, Write_to_User, and Respond_to_User. These procedures hide the method of access to the user and the syntax of communication with the user.

The Read_User_Command procedure reads a single line of text from the Debugger console of the 432 system and returns pointers to two string variables and their respective lengths; the Command, Command_Length, Arg_string, and Arg_string_Length. The command word, Command, is the first word of the input line delimited by a blank space. The remainder of the input line is placed in the Arg_string. Generally, this will contain the command arguments and options requested by the user. The lengths of these variables are provided to make processing the input easier.

The procedure Get_Argument takes Arg_string and Arg_string_Length as inputs and returns Argument and

Argument_Length to the calling program. The values of Arg_string and Arg_string_Length are modified to reflect that the Argument has been removed from the left end of the string. Get_Argument is used by the procedures within the System Commands package which require access to the individual arguments from the input line.

The procedures Read_from_User and Write_to_User are provided to allow access to the system operator by functions executing as a result of a previous command. Both procedures require a pointer to a string variable, Text_string, and an integer, Text_string_Length, as input and return the same variables modified by the operation performed.

The procedure Respond_to_User is a simple modification of Write_to_User. In addition to writing Text_string to the user console, the procedure also writes a line-feed character and a command prompt. The procedure is intended to be used upon completion of all shell commands, to write the response message to the console and prompt for the next command.

System Commands. The System Commands package contains the procedures which perform the requested commands. The package hides the implementation of the commands and even their names, which means, all modifications to the commands will affect only this package. Table 3-III lists the procedures found in the System

TABLE 3-III

Procedures of the System Commands Package

2.1 Perform_System_Command 2.1.1 Determine_Command 2.1.1.1 Set 2.1.1.2 Help 2.1.1.3 Copy 2.1.2 Create_Response

Commands package. The only System Commands procedure which may be called from outside the package is Perform_System_Command.

Perform_System_Command uses the procedure Determine_Command to select the command to be executed and call the command procedure to perform the function (see Figure 3-5). The output, from the command procedure, is a pointer to a text string, Response, and an integer value, Response_Length. A null pointer and length of zero are returned if the operation is successful. Otherwise, the string contains an error message. The exact message is determined from the status returned from the command procedure execution. The Create_Response procedure takes the status value as an input and returns Response and Response_Length. There are three commands implemented to demonstrate the I/O Interface. These are SET, HELP, and COPY.

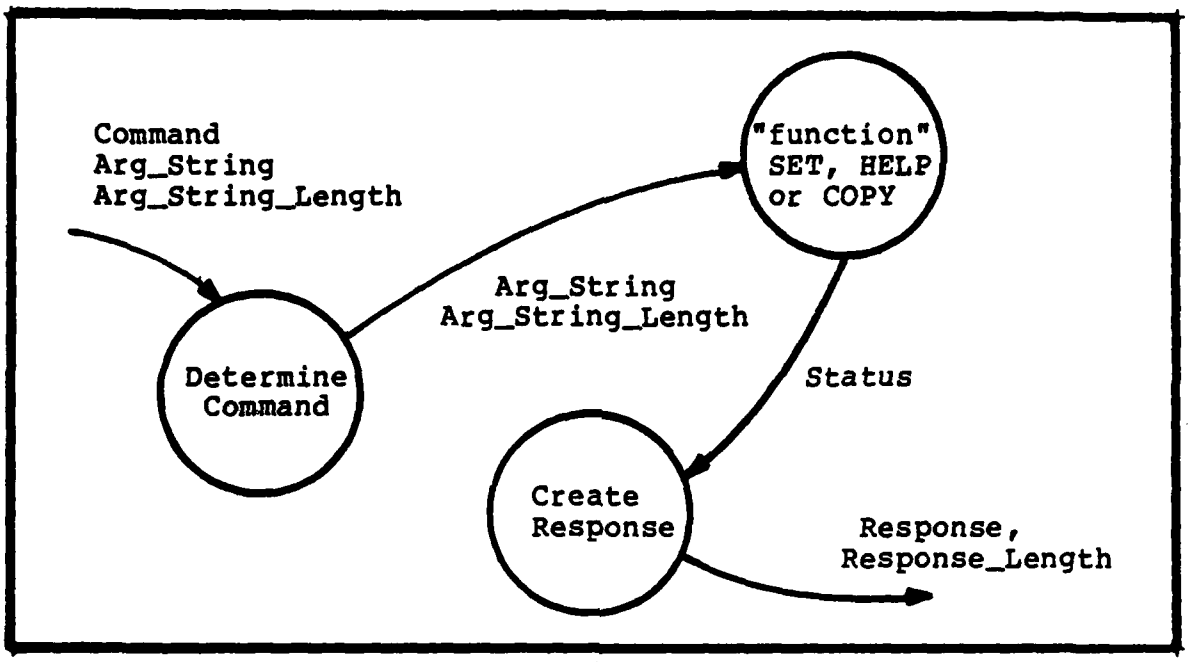


Figure 3-5. Perform_System_Command Data Flow

The SET command allows the user to modify system parameters during execution. The primary uses affecting the I/O Interface system are setting the time (used by the UAs to time stamp the messages) and setting default device naming for source and destination arguments. The latter allows the user to give the only the local device name and the system will attach the more significant members of the device name from the default selected.

The HELP command is implemented in a minimal form to remind the user of the commands available and their respective syntax and semantic requirements. The procedure implementing the command contains the text responses as constant values and returns a pointer to the proper text

string when that information is requested.

The COPY command demonstrates the use of the I/O Interface. This command calls the services of the User Shell Agent package to perform the information transfer requested. First, the source and destination files are opened. Then the source is read and the data is written to the destination. Finally, the source and destination devices are closed and a status is returned to the Perform_System_Command procedure. After each step (which is actually generating a CBMS message) the procedure waits for a reply message indicating the status of the operation. If an error is indicated the process performs the necessary operations to close any files and return an error status.

Device Abstraction. System devices are also users of the I/O Interface system. Unlike the User Shell, they will only respond when they receive messages from other system users. These devices are provided as a package of functions which establish the characteristics of the device. This abstraction of each device is a functional view of the services provided by the implementation of the device. Thus, the I/O Interface devices are implemented as a package of device procedures which call upon the device driver routines of the operating system to perform the function. Each device package also provides a test procedure which simply returns a valid status if the device is active and ready to receive messages. The device packages include a

TABLE 3-IV

Procedures of the Printer System Package

3.1 Ps_Open 3.2 Ps_Close 3.3 Ps_Print 3.4 Ps_Form_Feed 3.5 Ps_Title_Page 3.6 Ps_Test

Printer System (PS), the ISIS File System (IFS), and the Series III Console (S3C).

The Printer System provides six procedures for users of the printer device; Ps_Test, Ps_Open, Ps_Close, Ps_Print, Ps_Form_Feed, and Ps_Title_Page. These are listed again in Table 3-IV. The Ps_Open and Ps_Close are used to allocate and deallocate the use of the device, respectively. While the printer is in use, all messages from other users, except requests for Ps_Test) are not accepted. The Ps_Write procedure takes two inputs, Buffer_Access and Buff_Size, and calls the device driver to write the contents of the buffer to the attached device. The Ps_Form_Feed requires no inputs and simply generates the proper control signal for a formfeed by the device. Likewise, the Ps_Title_Page procedure generates a control for a formfeed, but, then creates a title page from information in the message and generates another formfeed control upon completion. All the PS procedures return a status value indicating the success

TABLE 3-V

Procedures of the ISIS File System Package

4.1 Ifs_Open
4.2 Ifs_Close
4.3 Ifs_Read
4.4 Ifs_Write
4.5 Ifs_Delete
4.6 Ifs_Rename
4.7 Ifs_Reset
4.8 Ifs_Test

or failure of the operation.

The ISIS File System provides services for file handling operations. Table V lists the procedures in the IFS package. These procedures include Ifs_Read, Ifs_Write, Ifs_Delete, Ifs_Rename, Ifs_Open, and Ifs_Close. All these procedures require a file name, File_Name and File_Name_Length, as input and return a status value for the operation. In addition, the Ifs_Read and Ifs_Write procedures require a pointer to a data buffer, Buffer_Access, and an integer length, Buff_Size, for the data. The Ifs_Rename procedure also requires a pointer to a second file name, New_Name, and its length, New_Name_Length. These procedures call ISIS operating system procedures to access the file device drivers. The Intellec Series III Microcomputer Development System Programmer's Reference Manual" (Ref 13) gives detailed information on access to these executive procedures. The IFS package can handle up

TABLE 3-VI

Procedures of the Series III Console Package

5.1 S3c_Read 5.2 S3c_Write 5.3 S3c_Clear_Screen 5.4 S3c_Test

to 4 files open simultaneously. The Ifs_Open and Ifs_Close procedures maintain a table of the files available for access at any time and the user allocated access to each file. Thus, commands may be given to transfer data between files or multiple users may access files simultaneously (obviously, not necessary for this single-user implementation of the User Shell system). The Ifs_Reset procedure closes all files without regard to the user allocation. Therefore, Caution must be used when requesting a device reset function.

The Series III Console device package contains only four procedures; S3c_Write, S3c_Read, S3c_Clear_Screen, and S3c_Test (see Table 3-VI). The S3c_Read and S3c_Write procedures require buffer information, as with the IFS and PS packages, and all procedures return status information. The S3c_Clear_Screen procedure generates the control input necessary to cause the console CRT screen to be cleared. All these procedures are implemented by calls to ISIS operating system procedures.

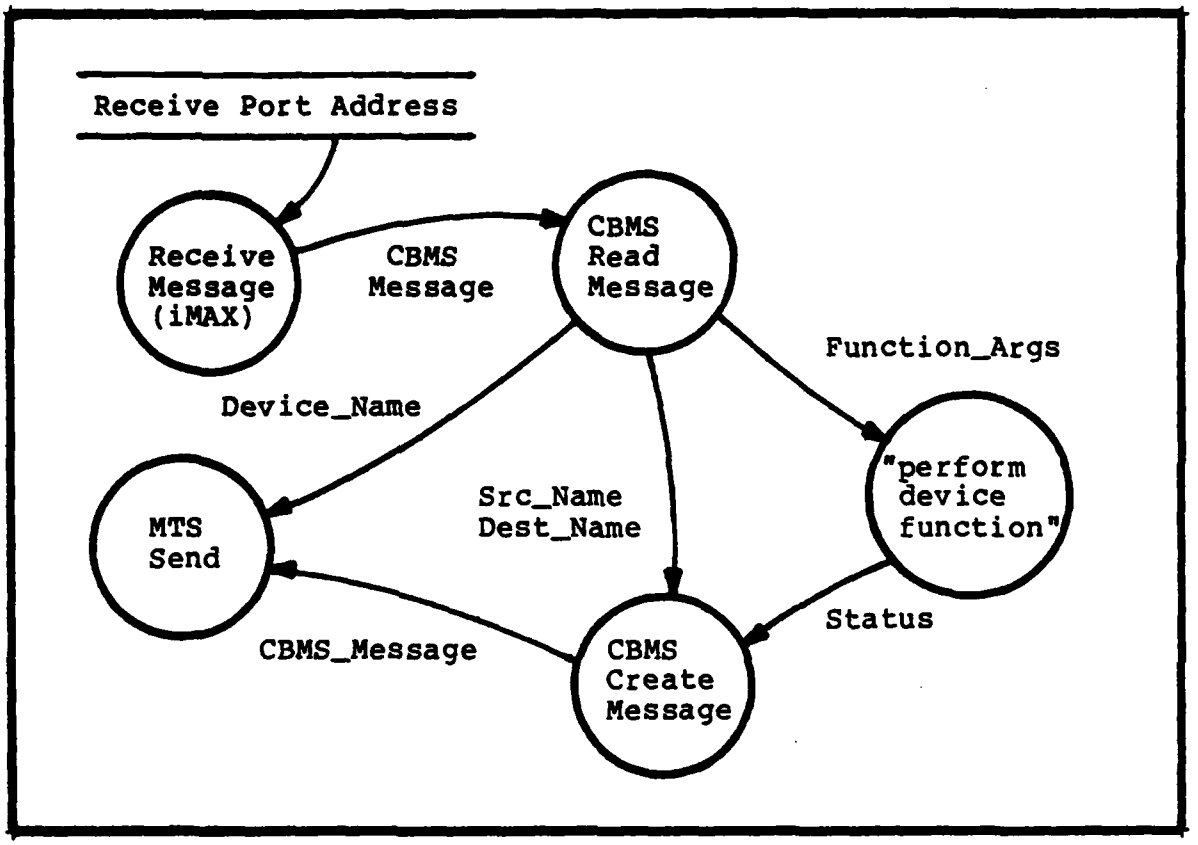


Figure 3-6. Typical User Agent Receive Procedure Data Flow

User Agent Sublayer.

Each User Agent package may be considered as two distinct sections. First, the Receive Section contains a single process which handles the messages sent to the UA. This process calls upon the services of the device abstraction to perform the function requested in the message. The following sequence of steps are performed in a continuous loop (see Figure 3-6):

1. Wait at the receiving port for a message to arrive.

2. Use the MTS sublayer procedures to read the message.

3. Use the User sublayer procedures to perform the function.

4. Use MTS sublayer procedures to create a reply message depending upon the status returned by the user or device procedure.

5. Send the reply to the MTS sublayer for routing to the originating UA.

The other part of the UA is the Send Section which contains the procedures called by the device or user to send messages to other devices. These procedures perform the following sequence of tasks (see Figure 3-7):

1. Use procedures of the MTS sublayer to create the message.

2. Send it to the proper UA.

3. Wait for a reply message indicating the result.

4. Return the operation status to the calling user or device.

In the AP, the UAs for each device are single sided systems. That is, they have only the Receive section of the User Agent package because they are "passive" users. An "active" user is capable of creating requests for action by other devices in the I/O Interface system. A "passive" user can only wait to receive messages and act upon the request. The User Shell, however, is an active user and its UA, User Shell Agent (USA), has both the Receive and Send sections.

There are two communications ports implemented with each UA. The first is the receive port where messages for

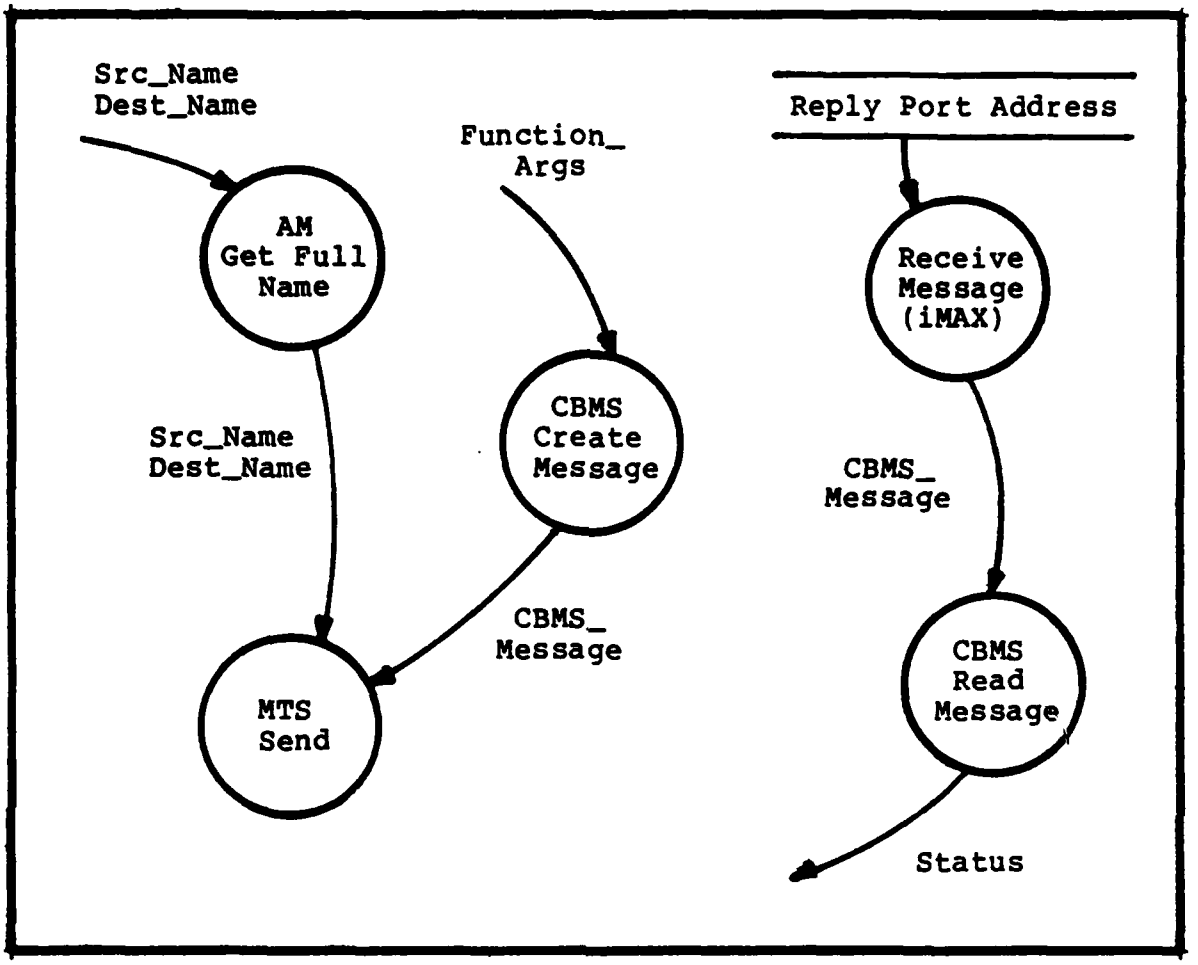


Figure 3-7. Typical User Agent Send Procedure Data Flow

that device are sent by the MTS. The second is the reply port where only the reply messages intended for that UA are sent. This double port system allows a user to send a message to itself or, in the case of the IFS, send information from one file to another. Without this system, a form of "dead lock" would occur when a user waits for a reply from itself. The alternative would be a second level of queues which would mean more processing overhead and

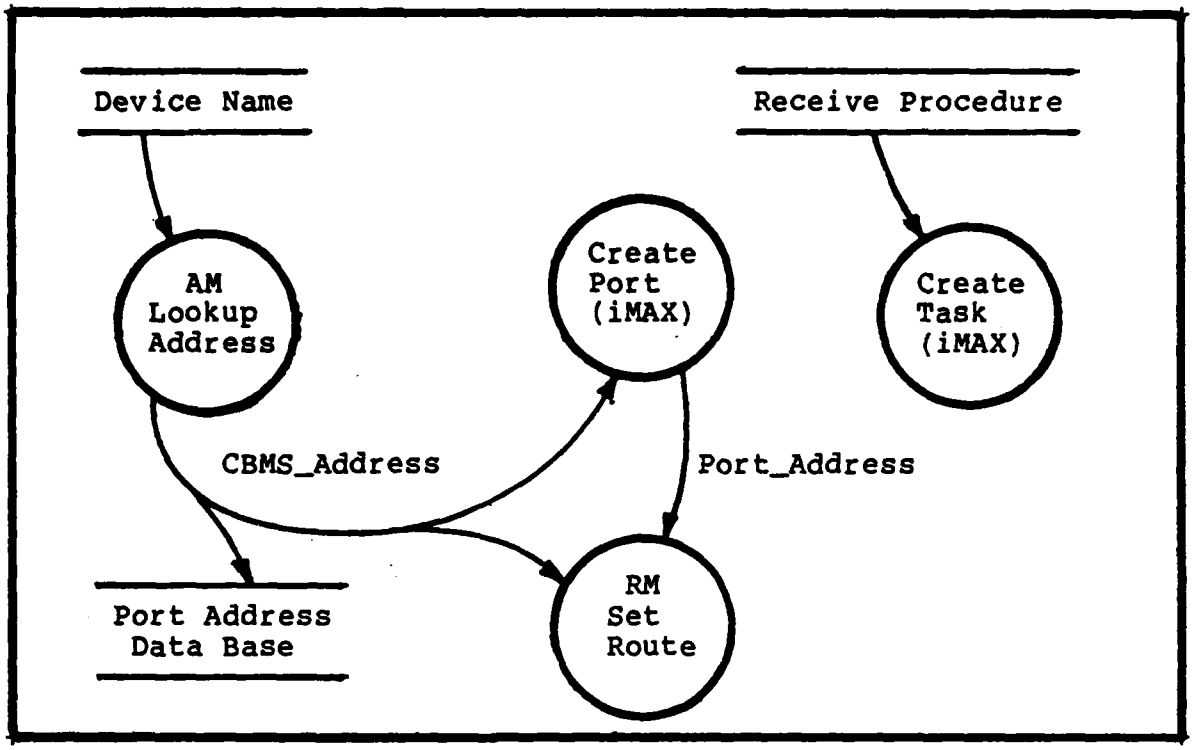


Figure 3-8. Typical User Agent Initialization Data Flow

almost twice the number of port implementations which would require large memory allocations (Ref 25:COM-2,SIZ-3).

Each UA package contains an initialization procedure which uses iMAX procedures to create the communication ports necessary for the user. The initialization procedure also starts the receiver process which handles incoming messages for the UA. Figure 3-8 shows the data flow in the initialization procedure for a typical UA. The process of creating the communications port would need to be done twice in an active UA having a receive port and a reply port.

TABLE 3-VII

Procedures of the User Shell Agent Package

6.1 Usa_Init
6.1.1 Usa_Receive
6.2 Usa_Open
6.3 Usa_Close
6.4 Usa_Read
6.5 Usa_Write
6.6 Usa_Page
6.7 Usa_Title
6.8 Usa_Delete
6.9 Usa_Rename
6.10 Usa_Reset
6.11 Usa_Get_Config
6.12 Usa_Set_Config
6.13 Usa_Test

User Shell Agent. The User Shell Agent package is the only complete UA implementation in the I/O Interface system on the 432 Micromainframe. In particular, this means that it contains the only implementation of the Send section of the UA. This section of the package contains a procedure for each I/O Interface function that is available to the user. All of these procedures require the destination device be designated by Device_Name and Device_Name_Length as inputs. The procedures also require inputs corresponding to the function requested. For example, the Usa_Test procedure requires no additional inputs, but, the Usa_Write and Usa_Read procedures needs information about a buffer of data, Buffer_Access and Buff_Size. Each of these procedures uses the MTS sublayer functions to send a message to the

TABLE 3-VIII

I/O Interface Replies to Function Requests

	I/O Interface Reply Codes							
	0	1	2	3	4	5	6	7
I/O Interface Function	Ok	Invalid Command	End of File	Bad Data	- Device - Error	Closed	Off	Busy
Open	x				x		x	x
Close	x				x		x	x
Read	x	x	x	x	x	x	x	x
Write	x			x	x	x	x	x
Page	x	x			x	x	x	x
Title	x	x		x	x	x	x	x
Delete	x	x			x		x	x
Rename	x	x		x	x		x	x
Reset	x	x			x		x	x
Get Config		x					x	
Set Config	x	x		x	x	x	x	x
Test	x				x		x	

device requested and then wait for a reply which indicates the status of the operation. Table 3-VIII shows the possible reply indications which can be received for each function request. The possible replies depend to some extent on the existence of a device which can perform the function requested. In the case of the Usa_Set_Config procedure, none of the devices implemented will actually allow a modification to their configuration. This is a limitation of the ISIS operating system and the Series III configuration (Ref 13:2-1/2-5). However, the Printer System device and the Series III Console device respond to the function with a valid status, as if the configuration were

TABLE 3-IX

Procedures of the Device Agent Packages

Printer System Agent Package 7.1 Psa_Init 7.1.1 Psa_Receive ISIS File System Agent Package 8.1 Ifsa_Init 8.1.1 Ifsa_Receive Series III Console Agent Package 9.1 S3ca_Init 9.1.1 S3ca_Receive

modified. This is done because there is nothing that would result from a change in configuration of these devices which would be necessary to the requesting user (User Shell). Only the device needs to know the actual configuration. Thus, the possible replies for a configuration request include "Ok" and "invalid command" (the file system does not change configuration). On the other hand, none of the devices will accept a request for configuration information (Usa_Get_Config). Therefore, the only reply possible is an indication of an invalid function request.

Device Agents. Device Agents are the UAs for the I/O Interface system devices. These are implemented on the AP system. Each device has its own UA or Device Agent. Therefore, there is a Device Agent for the Printer System, called the Printer System Agent (PSA), the ISIS File System,

TABLE 3-X

I/O Interface Function Mapping to System Devices

I/O Interface Function	Device Functions PS	IFS	S3C
Open	Open	Open	2
Close	Close	Close	2
Read	1	Read	Read
Write	Print	Write	Write
Page	Form Feed	2	Clear Screen
Title	Title Page	2	Clear Screen
Delete	1	Delete	1
Rename	1	Rename	1
Reset	Close	Reset	2
Get Config	1	1	1
Set Config	2	1	2
Test	Test	Test	Test

1 - Not Implemented, UA replies "command invalid"
2 - Not Used, UA replies "ok"

called the ISIS File System Agent (IFSA), and the Series III Console, called the Series III Console Agent (S3CA). The procedures in these packages are listed in Table 3-IX.

These Device Agents map the I/O Interface functions into the device functions of each device abstraction. For example, the I/O Interface "page" function is mapped to the Printer System procedure Ps_Form_Feed and the Series III Console procedure S3c_Clear_Screen but has no counterpart in the ISIS File System and would cause a reply message indicating an invalid function request for that device. Table 3-X shows a complete mapping of the I/O Interface

functions into each device abstraction.

Since the devices are passive users, their UAs contain only the Receive section of the UA. Each Device Agent has only one procedure which may be accessed by processes outside the package. This is the Init procedure. The purpose of this procedure is to initialize the communications port necessary for the UA to receive messages and start the Receive procedure which waits at the port for messages as shown in Figure 3-8 above. The Init procedure calls iMAX AP executive to create the port and then begin execution of the Receive process (Ref 25:IOI-5/IOI-7,IOI-29/IOI-35).

The Receive process is not executable by procedures outside the package. The procedure implements an abstraction of the device within the domain of the I/O Interface. That is to say, the Receive procedure properly handles all the functions defined by the I/O Interface protocol and generates a reply message indicating the results of the function. Depending upon the particular device, the procedure either calls a procedure from the User Sublayer implementation of the device or replies directly to the originator of a message requesting a function which cannot be performed by the device or user at that device agent.

Message Transfer Sublayer

The Message Transfer Sublayer is the lowest level of the

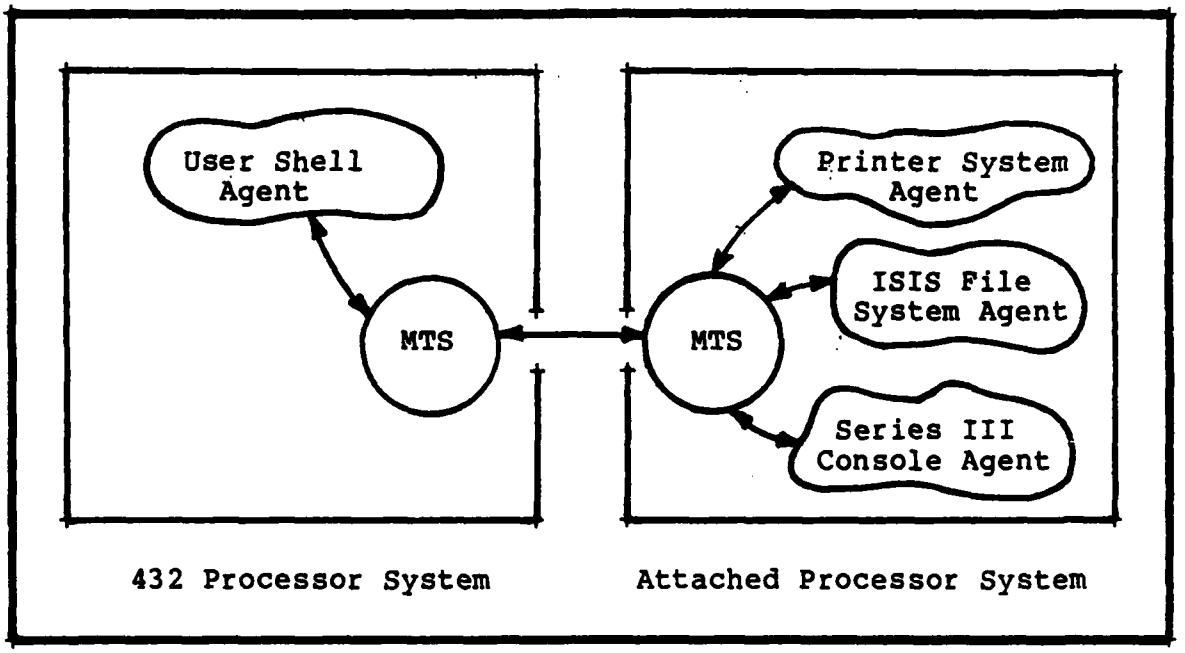


Figure 3-9. I/O Interface System Organization

I/O Interface system implementation. The sublayer exists in both the 432 processor system and the AP system. A functionally identical set of programs must exist in both systems. There are four packages which form the Message Transfer Sublayer; the Message Transfer System (MTS), the Address Manager (AM), the Route Manager (RM) and the Computer Based Message System Manager (CBMS). The Message Transfer System package contains the process for controlling the message movement among the local UAs and passing messages intended for remote UAs to the proper MTS port (see Figure 3-9). The Address Manager package contains the procedures for mapping the device names to CBMS addresses and the Route Manager provides the routing function by

TABLE 3-XI

Procedures of the Message Transfer System Package

10.1 Mts_Init 10.1.1 Mts_Receive 10.2 Mts_Send

mapping the CBMS addresses to the communications port where that device receives messages. The CBMS Manager package contains procedures for creating and accessing the information in CBMS messages with the I/O Interface format described in Chapter II. Each of the MTS sublayer packages is described further in the following sections.

Message Transfer System. The MTS package is similar to the UA packages and contains a send procedure and a receive procedure (see Table 3-XI). Like the UA initialization procedures, the MTS_Init procedure creates the port where the messages will be received and starts the procedure MTS_Receive (see Figure 3-10). The MTS_Receive procedure performs the following tasks in a continuous loop:

1. Wait to receive a message at the MTS port.
2. Use CBMS procedures to read the destination device name.
3. Use AM procedures to look up the address of the device.
4. Use RM procedures to look up the route for the device.
5. Send the message to the appropriate UA or MTS port.

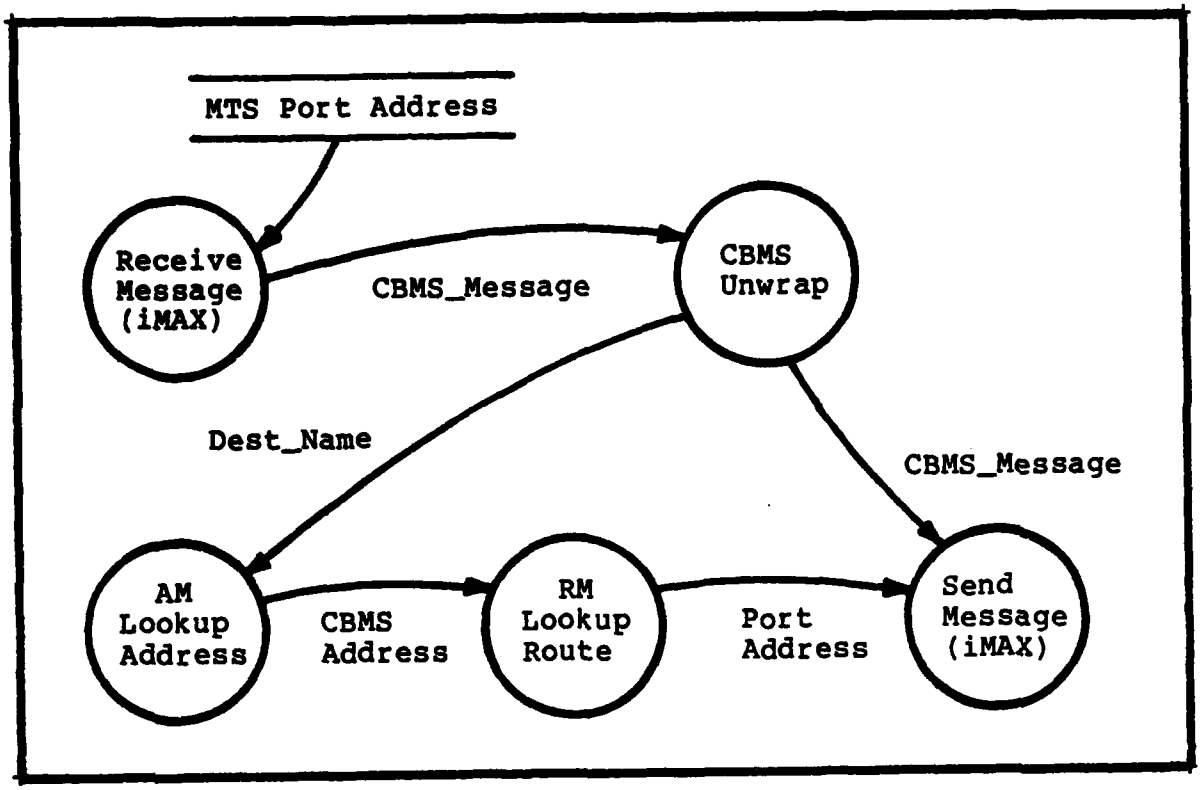


Figure 3-10. Mts_Receive Data Flow

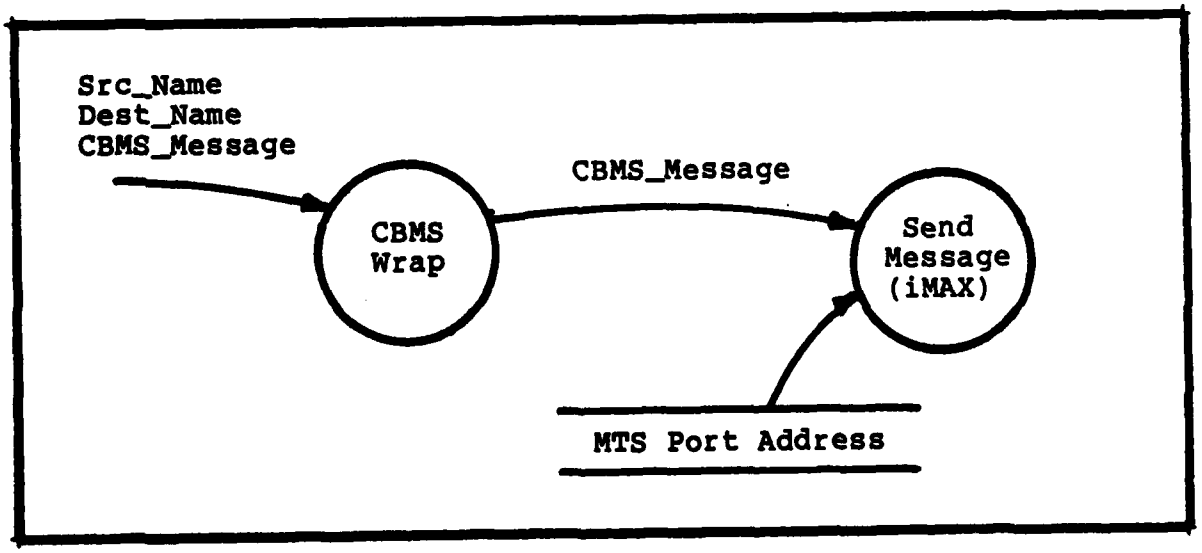


Figure 3-11. Mts_Send Data Flow

TABLE 3-XII

Procedures of the Address Manager Package

11.1 Am_Init 11.2 Am_Set_Default 11.3 Am_Lookup_Address 11.4 Am_Get_Full_Name

The send procedure, MTS_Send, requires the source and destination device names, Src_Name and Dest_Name, and access to a message. The procedure creates the CBMS message envelope, which contains the source and destination device names, and moves the message to the MTS receiving port (see Figure 3-11).

Address Manager. The Address Manager package provides services necessary to maintain a system for mapping device names to CBMS addresses. The four procedures in the Address Manager package are listed in Table 3-XII. The Am_Init procedure initializes the mapping structure with no entries. In this initial state, all requests for addresses would be returned with the status indicating that the device name is invalid. The Am_Lookup_Address procedure requires the name of the device, Device_Name, and its length, Device_Name_Length, and a pointer to storage for the CBMS_Address. The procedure returns with CBMS_Address pointing to the address for the device name given. The Am_Set_Default procedure accepts a Device_Name and its

length as input and stores them for use in creating the complete device name for a given reference by Am_Lookup_Address.

The Am_Get_Full_Name procedure takes a device name, entered by the user, and uses the previously defined Default_Name to determine the "complete" device name. The complete device name contains four parts; Country, Network, Host, and Device. In addition a file name may be specified as a fifth part of the source or destination device name.

The Country and Network portions of the name are each 4-character designations which, together, specify a node of the global network system. The 432 system would be connected to one port of such a node and, so, the Country and Network portions of the device name are constant. The I/O Interface implementation of the Address Manager package recognizes only one value for each of these; "RM67" for Country and "NET0" for Network. If additional systems need to be identified, the package can be modified, but, only the Address Manager package needs to be changed.

The Host and Device designators are 3-character names. The Host name specifies either the 432 processor ("432") or the Attached Processor ("MDS"). The Device portion of the name may indicate the User Shell on the 432 processor ("USR") or the Printer System ("PTR"), the Series III Console ("CON"), or the ISIS File System ("DSK") of the AP system. Figure 3-12 shows the relationships among the

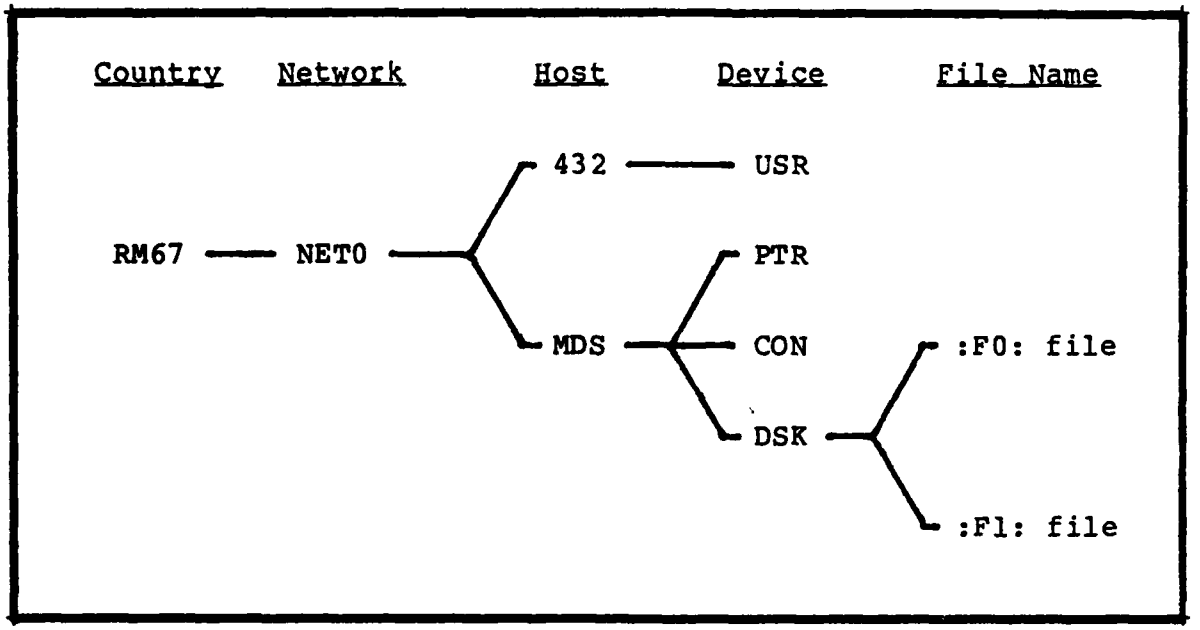


Figure 3-12. I/O Interface Device Naming Structure

device names.

The format for the name is also established within the Address Manager Package. Each of the major portions of the name is separated by a slash ("/"). Thus, the complete name for the I/O Interface Printer System device would be:

"RM67/NET0/MDS/PTR".

When necessary, a file name is separated from the device name by another slash. The structure of the file name is defined by the ISIS system user's manuals (Ref 13:2-3). So, the complete name for a file called "test.txt" on drive ":F0:" of the ISIS File System would be:

"RM67/NET0/MDS/DSK/:F0:TEST.TXT".

More examples of this format are given in the I/O Interface Users Manual (Appendix H).

TABLE 3-XIII

Procedures of the Route Manager Package

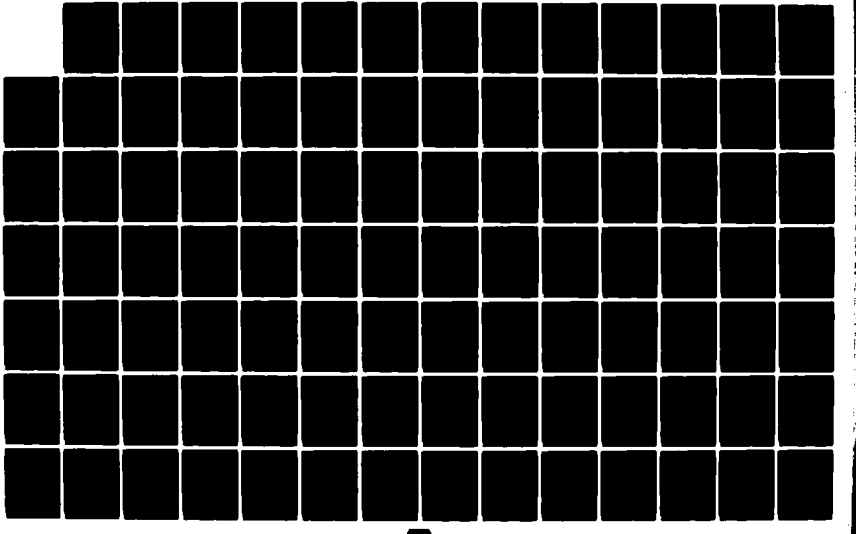
12.1 Rm_Init 12.2 Rm_Set_Route 12.3 Rm_Lookup_Route

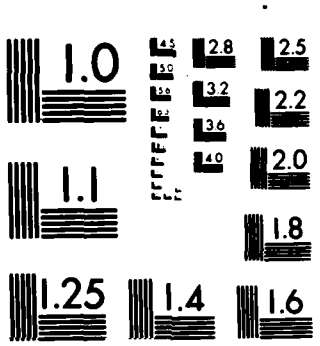
The actual implementation of the mapping structure is not accessable to any procedure outside the AM package. Thus the package could implement a tree structure to provide address mapping (in a larger system, this may be more efficient), however, for the present configuration of the Intel 432 system, a simple table structure is adequate. The size of the table is a constant in the package. Therefore, any increase in the number of entries (system devices) requires a modification of the code.

Route Manager. The Route Manager package provides access to the data structure which maps the CBMS address of a device to the next communications port which should receive the message. The procedures in the package are listed in Table 3-XIII. The Rm_Init procedure initializes the data structure to have no entries. The Rm_Set_Route and Rm_Lookup_Route procedures must have complete device names (from the Am_Get_Full_Name procedure) for proper referencing. The Rm_Set_Route procedure is called by each UA, during its initialization, with the CBMS address and

AD-A138 429 DESIGN AND IMPLEMENTATION OF AN INPUT/OUTPUT INTERFACE PROTOCOL FOR THE I..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. K N COLE

UNCLASSIFIED DEC 83 AFIT/GE/EE/83D-17 F/G 17/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

TABLE 3-XIV

Procedures of the CBMS Manager Package

13.1 Cbms_Create_Message 13.2 Cbms_Read_Message 13.3 Cbms_Wrap_Message 13.4 Cbms_Unwrap_Message

communications port address to be entered into the structure. The Rm_Lookup_Route procedure is used by the Mts_Receive procedure to determine the next communications port for a message.

Like the Address Manager data structure, the Route Manager structure is not directly accessed by any routines outside the package. The internal structure of the data is a simple table of a fixed size. However, unlike the Address Manager system, the table of routes is not complete. For any address that represents a device on another host, there is always the same route, the MTS port of the other machine.

CBMS Manager. The CBMS Manager package provides procedures necessary to manipulate the I/O Interface message structures. The implementation of the CBMS messages, as described in Chapter II, is known only by the procedures of this package (see Table 3-XIV). The Cbms_Create_Message procedure requires input arguments with all the information to be placed in the message. After using iMAX procedures to create the message data structure, the message fields are

created from the input data. The opposite function is performed by the Cbms_Read_Message procedure which destroys the message structure and provides the data found in the message fields to the calling routine. The Cbms_Wrap and Cbms_Unwrap procedures are used by the MTS package procedures and place the Message Transfer System "envelope" on the message. The envelope is simply the source and destination user names placed at the head of the message string. To simplify the handling of the message structures, the space for the envelope data is included in the CBMS message structure allocated when the message is created.

Summary

This chapter has presented the design of the I/O Interface for the 432 Micromainframe Computer System. The three sublayers within the ISO Applications Layer of protocol create an efficient organization for software design. The implementation details of the structures required for each sublayer do not cross protocol layer boundaries. The object-oriented use of packages within each sublayer has further protected access to the data structures. Use of the heirarchial and object-oriented design techniques has created a system organization that is maintainable and flexible. With completion of this interface system, the 432 can communicate with the outside world in an organized manner. Future designers can build upon this system of input and output functions.

IV. SYSTEM TESTING

Introduction

The first three chapters have presented the objectives, requirements, and design for the I/O Interface. The purpose of this chapter is to define the testing procedures needed to validate the software developed to meet these objectives, requirements and design goals of the interface on the Intel 432 Micromainframe computer system.

In general, testing proceeds concurrently with the implementation of the requirements and design. Thus, the testing sequence follows the top-down approach of the design method. The CBMS and ISO models, described in Chapters II and III, create a distinct hierarchy within the design, which provides natural divisions for software validation. For example, the operation of each CBMS sublayer, within the ISO Applications Layer, can be tested as it is added to the system (i.e., first the User Sublayer, then the User Agent Sublayer, and finally, the Message Transfer Sublayer).

Test Design Structure

The design of the I/O Interface for the 432 Micromainframe implements each of the CBMS sublayers on the 432 processor and the Attached processor system (see Figure 4-1). The 432 processor software is written in Ada language and its execution, for testing, is controlled by the

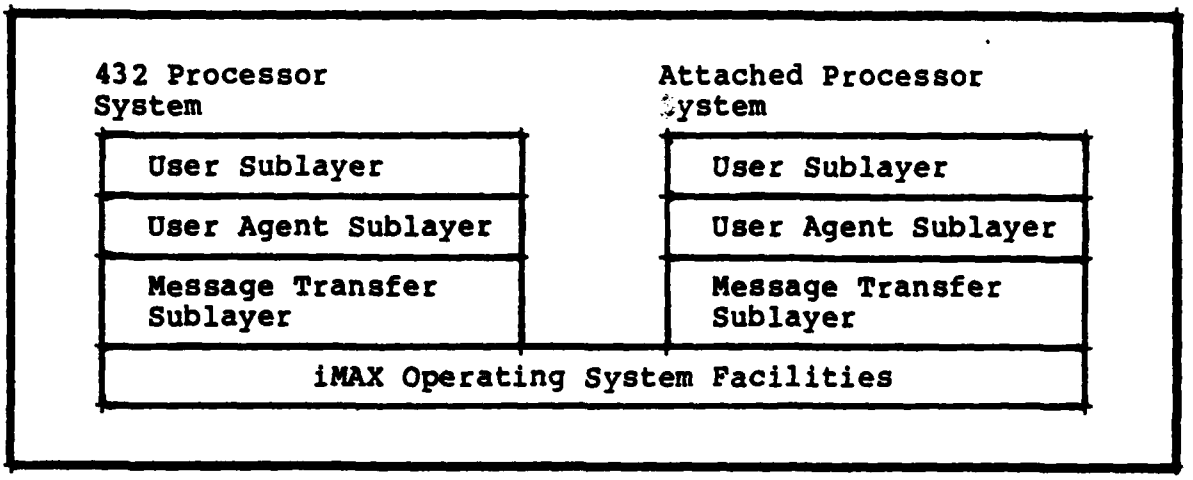


Figure 4-1. I/O Interface Software System Structure

Debugger system (Intel Series III MDS) connected to the 432 processor hardware. The Attached processor software is written in PLM-86 and executes directly on the Intel Series III MDS. Testing procedures can be used to validate each processors software individually and then integrate the two systems to validate the interconnection hardware and software. Thus, the testing procedures follow the hardware and software structures of the system.

The testing procedures are structured to validate each sublayer of the CBMS hierarchy, individually. This is, essentially, the "black-box" testing technique (Ref 38:86,311). Each level of the hierarchy is tested as a distinct system which accepts a defined set of inputs and responds in an expected manner (i.e., each level performs certain tasks for the system).

Testing at a lower level, within the sublayer, is left

to the code development process during the top-down implementation of the modules. The modular and object-oriented structure of the system software ensures that the only coupling occurs at the functional level of the sublayer boundaries. In other words, invalid code within a sublayer can only affect the data structures of that sublayer. Thus, thorough testing of the inputs and outputs of each sublayer is sufficient to validate the operation of the modules within the sublayer.

The overall complexity of the 432 system environment dictates one additional test be performed. As described in Chapter II, the Cross Development System for the 432-Ada software involves two computer systems in addition to the 432 Micromainframe computer; the VAX 11/780 Host system and the Intel Series III MDS Debugger system. A test to validate this development environment would increase confidence in the system. For the 432 CDS environment, the most direct test involves taking an existing program, known to be correct, and performing tasks usually done during software development (i.e., compile the source, link the object modules, and run the program in a test environment). This environment test is done, before any I/O Interface system testing, to validate the interconnection and software utilities of the 432 Cross Development System.

There are also two other reasons for beginning the the testing process at this point. Since the original

TABLE 4-I

I/O Interface Testing Procedures

1. Environment Validation Test

2. 432 Processor System Validation
   - 2.1 User Sublayer Test
   - 2.2 User Agent Sublayer Test
   - 2.3 Message Transfer Sublayer Test

3. Attached Processor System Validation
   - 3.1 User Sublayer Test
   - 3.2 User Agent Sublayer Test
   - 3.3 Message Transfer Sublayer Test

4. System Integration Test

installation of the 432/670 system, the 432 and both Series III Development Systems had been physically relocated in the building. This required re-routing of the hard-wire connection to the VAX system. Also, a new release of the hardware (designated Release 2.1) had been received with improved versions of the software, as well. This new configuration of both hardware and software had not been used. Therefore, it is necessary to perform this basic test of the system to establish confidence in the configuration, before software development begins.

The remainder of this chapter is organized according to the testing procedure outline given in Table 4-I. The methods and expected results for each test are discussed in the following sections:

TABLE 4-II

Environment Validation Test Procedure

Performed on VAX 11/780:

1) Verify the proper environment file names. (VMS or UNIX system editor)

2) Compile the source files. (IDA)

3) Link the resulting object modules with the iMAX operation system module. (LINK432)

Performed on Series III MDS:

4) Download the executable file to the Series III MDS workstation. (DNLOAD)

5) Verify the hardware configuration of the 432 Micromainframe Computer. (DSP432)

6) Load and execute the PRIME program. (DEB432)

Environment Validation Test

The method used for this test is to take a program, that is known to be valid, through the complete development cycle. The PRIME program is an Ada language program which computes prime numbers. This program was supplied by Intel as an example software system which would execute on the 432 system (Ref 21:G-4/G-7). The test method, then, is to use the utilities of the development system environment to compile, link, and execute the PRIME program.

Table 4-II lists the steps in the test procedure. The names of the software utilities are given in parentheses with a reference to the manual describing its use. The Intel 432 Cross Development System (CDS), including the operation of its utilities, is discussed in more detail in Appendix H.

The expected results are, simply, that the PRIME program executed properly on the 432 system. This validates that the 432's program development environment is functional and produces executable code for the 432/670 computer system.

To perform this test, the Intel 432/670 system should be configured as shown in Figure 4-2. This hardware environment includes only the Debugger and the 432 Processor systems. The PRIME program uses the system Debugger as the I/O device and, therefore, does not require another AP system. In fact, any unnecessary hardware in the system may cause the program to fail or act in an unpredictable manner.

The Ada language source files for the PRIME program are listed in Table 4-III. This list does not include the operating system packages required to create an executable code module. Refer to the Intel 432/670 Computer System Users Guide for a complete list of the software (Ref 35:Appendix B) and a tutorial on the operation of the PRIME program (Ref 35:Chapter 11).

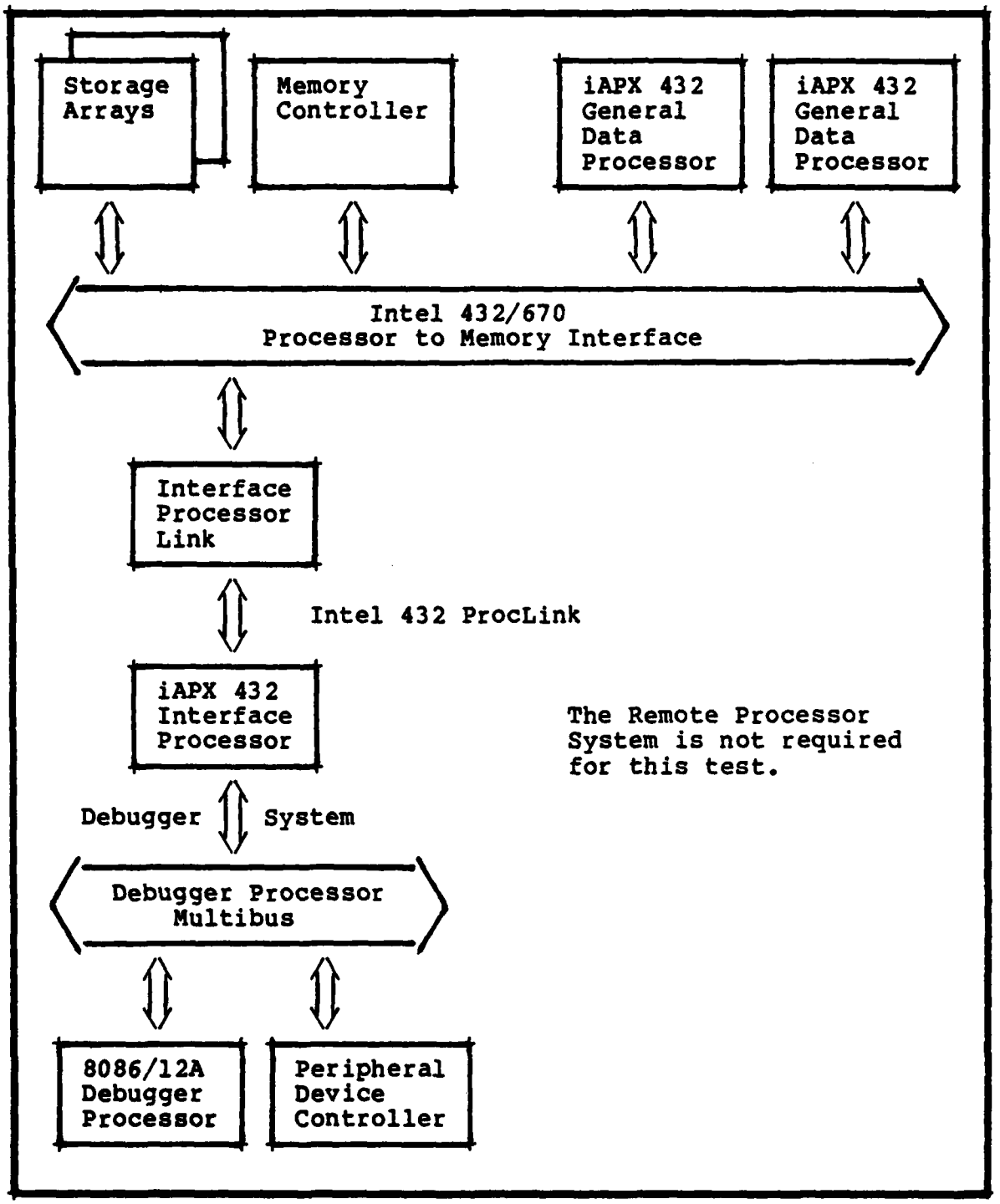


Figure 4-2. Environment Validation Test Hardware Configuration

TABLE 4-III

PRIME Program Software
(Ref 35:108)

Contents	File Name	
PRIME Example Program	[INTEL2.ACS.PRIME]	directory
TEXTIO routine specification	INTIO.MSS	
Console prompt routine spec.	PROMPT.MSS	
Console prompt routine body	PROMPT.MBS	
Primary program function	ISPRIM.MCS	
Main program control spec.	MAIN.MSS	
Main program control body	MAIN.MBS	
Process initialization spec.	PSERP.MBS	
Program linker directives	ISPRIM.LKD	

432 Processor System Validation

The I/O Interface software is validated by testing each of the three sublayers, in turn. Dummy modules are used for the procedures that are part of a lower sublayer and, therefore, not included in the test. Each dummy module writes a message to the Debugger console, indicating that the module has been called, and returns to the calling procedure.

The test is performed by executing each of the system commands (SET, HELP, and COPY). Table 4-III lists a minimum set of commands which may be used. In addition to these commands, a number of invalid commands should be entered to test the error handling properties of the system.

The hardware configuration for these tests is exactly the same as for the environment validation test, shown in

TABLE 4-IV

432 Processor Software Validation Test
Commands List

```
HELP
HELP SET
HELP COPY
SET DEFAULT RM67/NET0
COPY /432/USR /MDS/CON
COPY /MDS/CON /432/USR
COPY /432/USR /432/USR
SET DEFAULT RM67/NET0/MDS
COPY /DSK/:F1:TEST.TXT /PTR
```

Figure 4-2). The Attached Processor system is not required for these tests because no messages will be sent to other hosts in the network.

The following sections describe the software required and the results expected from the commands in Table 4-IV, for each sublayer:

User Sublayer Test (432). The first sublayer in the CBMS hierarchy is the User Sublayer. This sublayer of the I/O Interface has only one entity on the 432 processor; the User Shell. The software files required for this test are listed in Table 4-V. Source code listings of the test file (Ustest.mbs) and other 432 processor files can be found in Volume II of this report.

Without any support from the lower sublayers, the shell can only respond completely to commands which only involve itself. That is, the HELP command is completely functional,

TABLE 4-V

User Sublayer Validation Software (432)

File Name	File Contents
Cbms.mss	CBMS Manager Specification
Rm.mss	Route Manager Specification
Am.mss	Address Manager Specification
Mts.mss	Message Transfer System Spec.
Usa.mss	User Shell Agent Specification
User.mss	User Interface Specification
User.mbs	User Interface Body
Syscmd.mss	System Commands Specification
Syscmd.mbs	System Commands Body
Shell.mss	User Shell Specification
Shell.mbs	User Shell Body
Ustest.mbs	User Sublayer Test Modules Body

but, the SET and COPY commands cause flag messages to be printed by the dummy modules they call.

The operations of all the system shell commands are documented in Appendix H, including listings of the HELP command responses. The SET command responds with a flag message indicating that the procedure AM_Set_Default has been called from the Message Transfer Sublayer. The COPY command causes the following list of dummy procedures to be called in the User Agent Sublayer:

1. Usa_Open (open the source)
2. Usa_Open (open the destination)
3. Usa_Title (write title to destination)
4. Usa_Read (read from the source)
5. Usa_Write (write to the destination)

6. Usa_Close (close the destination)

7. Usa_Close (close the source)

The flag messages should appear in this order to indicate proper operation of the COPY function. Note that the source file is only read once because the dummy module for the Usa_Read procedure indicates an end-of-file condition on the first read action. Proper responses to the commands validates operation of the User Shell, User Interface, and System Commands packages of the User Sublayer.

User Agent Sublayer Test (432). The User Agent Sublayer, like the User Sublayer, contains only one entity on the 432 processor; the User Shell Agent. For this test, the User Sublayer test file is replaced by the software modules of the User Agent and a different set of dummy procedures in the User Agent test file (Uastest.mbs). Table 4-VI lists the files containing the User Agent Sublayer modules. Again, the source code listings are provided in Volume II of this thesis.

In the previous test, this sublayer generated the flag messages of the COPY command test. The HELP command responds completely, as in the test above. The COPY and SET commands will still not perform completely, however, without the Message Transfer Sublayer implementation.

The SET command responds exactly as before, calling the procedure AM_Set_Default. The COPY command, now, causes six

TABLE 4-VI

User Agent Sublayer Validation Software (432)

File Name	File Contents
Cbms.mss	CBMS Manager Specification
Cbms.mbs	CBMS Manager Body
Rm.mss	Route Manager Specification
Am.mss	Address Manager Specification
Mts.mss	Message Transfer System Spec.
Usa.mss	User Shell Agent Specification
Usa.mbs	User Shell Agent Body
User.mss	User Interface Specification
User.mbs	User Interface Body
Syscmd.mss	System Commands Specification
Syscmd.mbs	System Commands Body
Shell.mss	User Shell Specification
Shell.mbs	User Shell Body
Uastest.mbs	User Agent Sublayer Test Body

calls to the dummy procedure MTS_Send, indicated by flag messages. Each call to this procedure causes a reply message to be sent to the User Shell Agent reply port, allowing the calling procedure to continue processing. Again, a request to read a file will generate only one read-write operation, because the reply to this request indicates an end-of-file condition. This level of testing indicates the successful operation of the communication ports and message handling procedures on the 432 processor system. The next test completes validation of the 432 processor software as a stand-alone system.

The CBMS Manager package, of the Message Transfer Sublayer, must be implemented for this test. This package

TABLE 4-VII

Message Transfer Sublayer Validation Software (432)

File Name	File Contents
Cbms.mss	CBMS Manager Specification
Cbms.mbs	CBMS Manager Body
Rm.mss	Route Manager Specification
Rm.mbs	Route Manager Specification
Am.mss	Address Manager Specification
Mts.mss	Message Transfer System Spec.
Mts.mbs	Message Transfer System Body
Usa.mss	User Shell Agent Specification
Usa.mbs	User Shell Agent Body
User.mss	User Interface Specification
User.mbs	User Interface Body
Syscmd.mss	System Commands Specification
Syscmd.mbs	System Commands Body
Shell.mss	User Shell Specification
Shell.mbs	User Shell Body
Mtstest.mbs	Message Transfer Sublayer Test Body

contains the procedures for creating and accessing the I/O Interface messages. These functions are also validated during this test.

Message Transfer Sublayer Test (432). The last sublayer to be tested on the 432 processor contains the packages for translating names and addresses determining routes for message transfer, and moving messages among the I/O system agents. For this test, the previous test file is replaced by the modules of the Message Transfer Sublayer. There is, however, still a need for a test file of special software.

The messages cannot be sent to devices outside the 432 processor without having the Attached Processor system operational. To allow the Message Transfer Sublayer to be validated without the AP system, the test file (Mtstest.mbs) contains a modified version of the Address Manager package which maps all device names into the User Shell console. The test file also contains the Mts_init procedure which initializes the software package without creating the ports for communication with the missing AP system. Table 4-VII lists all the 432 software files necessary for the Message Transfer Sublayer validation.

The SET and HELP commands are completely functional in this test. The proper operations of both these commands are described in Appendix H. The COPY command performs the complete data transfer when demonstrated using the User Shell as the source and destination (complete designation "RM67/NET0/432/USR"). In addition, if any other valid device name is used for source or destination, it is handled as if the User Shell were the device requested. In other words, all valid device names are mapped into the User Shell by a modified Address Manager Package in the Message Transfer Sublayer.

This level of testing validates the data transfer, device naming, device addressing, message format, and message routing mechanisms of the I/O Interface on the 432 processor system. The interprocessor communication,

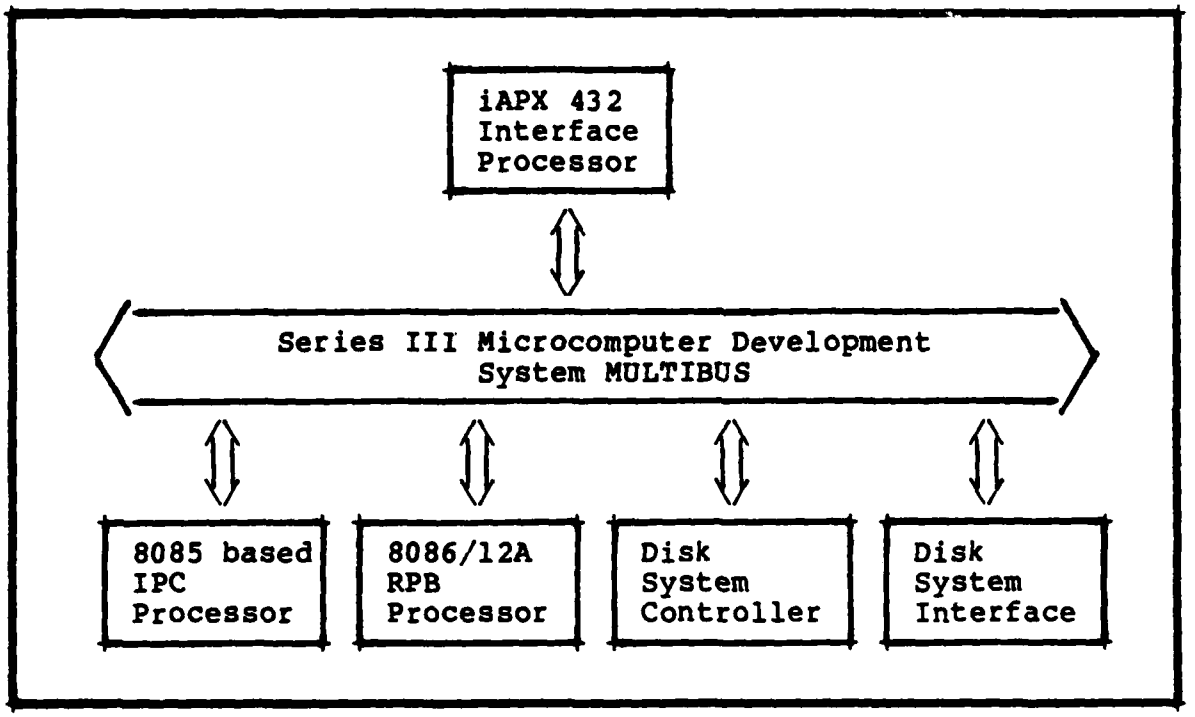


Figure 4-3. Attached Processor Software Validation Test Hardware Configuration

however, cannot be validated until the Attached Processor system is tested and the systems are physically connected.

Attached Processor System Validation

Validation of the AP software is more difficult than for the 432 processor system, because there is no controlling source device designed on the AP. The "commands" to the device abstractions come from messages sent from the 432 processor. To allow testing on the AP system, a "test shell" is included in the test software package. The test shell accepts commands from the Series III Console and calls procedures in the appropriate

TABLE 4-VIII

Attached Processor Test Shell Commands

Test Shell Command		Device Functions PS	IFS	S3C
0	Open	Open	Open	2
1	Close	Close	Close	2
2	Read	1	Read	Read
3	Write	Print	Write	Write
4	Page	Form Feed	2	Clear Screen
5	Title	Title Page	2	Clear Screen
6	Delete	1	Delete	1
7	Rename	1	Rename	1
8	Reset	Close	Reset	2
9	Test	Test	Test	Test
G	Get Config	1	1	1
S	Set Config	2	1	2

1 - Not Implemented, UA replies "command invalid"
2 - Not Used, UA replies "ok"

sublayer. These commands are shown in Table 4-VIII and the modules are listed in Volume II, with the dummy procedure modules for each sublayer. In general, the dummy procedures simply write flag messages to the Series III Console and return.

The hardware for the AP validation tests is, simply, the Intel Series III MDS. The configuration of the system is shown in Figure 4-3. Additional hardware in the system will, generally, not affect the operation of the AP software. However, it is important to ensure that enough memory is present to support the software requirements. The

TABLE 4-VIII

Attached Processor Test Shell Commands

Test Shell Command		Device Functions PS	IFS	S3C
0	Open	Open	Open	2
1	Close	Close	Close	2
2	Read	1	Read	Read
3	Write	Print	Write	Write
4	Page	Form Feed	2	Clear Screen
5	Title	Title Page	2	Clear Screen
6	Delete	1	Delete	1
7	Rename	1	Rename	1
8	Reset	Close	Reset	2
9	Test	Test	Test	Test
G	Get Config	1	1	1
S	Set Config	2	1	2

1 - Not Implemented, UA replies "command invalid"
2 - Not Used, UA replies "ok"

sublayer. These commands are shown in Table 4-VIII and the modules are listed in Volume II, with the dummy procedure modules for each sublayer. In general, the dummy procedures simply write flag messages to the Series III Console and return.

The hardware for the AP validation tests is, simply, the Intel Series III MDS. The configuration of the system is shown in Figure 4-3. Additional hardware in the system will, generally, not affect the operation of the AP software. However, it is important to ensure that enough memory is present to support the software requirements. The

TABLE 4-IX

User Sublayer Validation Software (AP)

File Name	File Contents
Cbms.inc	CBMS Manager Specification
Rm.inc	Route Manager Specification
Am.inc	Address Manager Specification
Mts.inc	Message Transfer System Spec.
Psa.inc	Printer System Agent Spec.
Ifsa.inc	ISIS File System Agent Spec.
S3ca.inc	Series III Console Agent Spec.
Ps.inc	Printer System Specification
Ps.plm	Printer System Body
Ifs.inc	ISIS File System Specification
Ifs.plm	ISIS File System Body
S3c.inc	Series III Console Specification
S3c.plm	Series III Console Body
Ustest.plm	User Sublayer Test Body

program space requirements are printed, on the output listing, by the PL/M-86 compiler.

User Sublayer Test (AP). In the AP, the User Sublayer contains the device abstractions for the three system devices; the Series III Console, the Printer System, and the ISIS File System. The PL/M-86 source files, containing the implementations of these abstractions, are listed in Table 4-IX. Source listings of these files and the User Sublayer test file (Ustest.plm) are provided in Volume II.

For this test, the test shell directly calls the functions of each device abstraction. The shell accepts only single character commands and performs the device functions as listed in Table 4-VIII. Each command prompts

the console operator for a device name (PTR, CON, or DSK) and, if necessary, for an ISIS compatible file name (Ref 13:2-1) or text data input.

This test validates the operation of each device and the performance of the functions which define the device abstraction used by the I/O Interface. The devices are expected to respond correctly to each command. In Addition, each command executed by a device generates a flag message indicating the procedure Send_Reply, in the User Agent Sublayer, has been called. Refer to Table 4-VIII to determine the functions applicable to each device. An invalid command selection for a device is ignored.

User Agent Sublayer Test (AP). The validation of the User Agent Sublayer is accomplished in the same manner, using the test shell, which, for this test, generates CBMS messages and sends them to the appropriate User Agent for the device requested. Table 4-X lists the source files required for the User Agent Sublayer Validation Test. Source code listings appear in Volume II of this thesis.

At this level, all the test shell commands listed in Table 4-VIII may be used with each device. The User Agent software handles the requests that are inappropriate for each device, using the CBMS Manager package to read the message contents. The CBMS Manager package must be implemented to allow the User Agent software to access the data in the shell command messages. The other modules of

TABLE 4-X

User Agent Sublayer Validation Software (AP)

File Name	File Contents
Cbms.inc	CBMS Manager Specification
Cbms.plm	CBMS Manager Body
Rm.inc	Route Manager Specification
Am.inc	Address Manager Specification
Mts.inc	Message Transfer System Spec.
Psa.inc	Printer System Agent Spec.
Psa.plm	Printer System Agent Body
Ifsa.inc	ISIS File System Agent Spec.
Ifsa.plm	ISIS File System Agent Body
S3ca.inc	Series III Console Agent Spec.
S3ca.plm	Series III Console Agent Body
Ps.inc	Printer System Specification
Ps.plm	Printer System Body
Ifs.inc	ISIS File System Specification
Ifs.plm	ISIS File System Body
S3c.inc	Series III Console Specification
S3c.plm	Series III Console Body
Uastest.plm	User Agent Sublayer Test Body

the MTS sublayer, used by the User Agent modules, generate flag messages for the shell console indicating their use. Thus, each command from the test shell performs a function on the device requested. The attempt to send a reply message, upon function completion, generates a flag message from the dummy procedure MTS_Send.

This test validates the operation of the User Agent software for each device and the CBMS Manager software which allows access to CBMS messages. The format syntax of the CBMS messages is also tested because the test shell uses the CBMS Manager software to create messages which are then read

TABLE 4-XI

Message Transfer Sublayer Validation Software (AP)

File Name	File Contents
Cbms.inc	CBMS Manager Specification
Cbms.plm	CBMS Manager Body
Rm.inc	Route Manager Specification
Rm.plm	Route Manager Body
Am.inc	Address Manager Specification
Mts.inc	Message Transfer System Spec.
Mts.plm	Message Transfer Body
Psa.inc	Printer System Agent Spec.
Psa.plm	Printer System Agent Body
Ifsa.inc	ISIS File System Agent Spec.
Ifsa.plm	ISIS File System Agent Body
S3ca.inc	Series III Console Agent Spec.
S3ca.plm	Series III Console Agent Body
Ps.inc	Printer System Specification
Ps.plm	Printer System Body
Ifs.inc	ISIS File System Specification
Ifs.plm	ISIS File System Body
S3c.inc	Series III Console Specification
S3c.plm	Series III Console Body
Mtstest.plm	Message Transfer Sublayer Test Body

by the User Agents.

Message Transfer Sublayer Test (AP). The final test of the AP system validates the operation of the addressing and routing mechanisms of the Message Transfer Sublayer. Table 4-XI lists the files necessary for operation of the Message Transfer Sublayer Validation Test. These source files are all listed in Volume II.

The test shell for this level generates messages which are given to the MTS_Send routine for entry into the MTS

TABLE 4-XII

System Integration Test Commands

```
HELP
HELP SET
HELP COPY
SET DEFAULT RM67/NET0
COPY /432/USR /MDS/CON
COPY /432/USR /MDS/PTR
COPY /432/USR /MDS/DSK/:F0:TEST.TXT
COPY /432/USR /MDS/DSK/:F1:TEST.TXT
COPY /MDS/CON /MDS/DSK/:F0:TEST2.TXT
SET DEFAULT RM67/NET0/MDS
COPY /DSK/:F1:TEST.TXT /PTR
COPY /DSK/:F0:TEST2.TXT /DSK/:F1:TEST.TXT
COPY /CON /PTR
COPY /CON /DSK/:F0:TEST3.TXT
COPY /DSK/:F0:TEST3.TXT /CON
COPY /DSK/:F1:TEST.TXT /CON
SET DEFAULT RM67/NET0
COPY /MDS/DSK/:F1:TEST.TXT /432/USR
```

receiving port. As with the 432 processor testing, the Address Manager package must be modified to send all messages intended for non-local devices to the local console (the Series III Console).

System Integration Test

The final test of the I/O Interface system is the System Integration Test. This test is intended to validate the operation of the complete system as a whole and is performed with no dummy software modules. The hardware and software environments, for this test, are described in Appendix H, with the I/O Interface operating instructions. The source files are listed in Volume II.

The test method is by execution of each of the system commands (SET, HELP, and COPY) with a range of arguments that validates the correct operation of each command and the interface with each device available in the system. Table 4-XII lists a minimum set of commands which may be used. In addition to this set, a number of invalid commands should be entered to test the error handling properties of the system.

The expected results of each command should be validated before entering the next command. Validation, in the case of file manipulations, may require halting the execution of the program and using the operating system utilities of the Series III MDS to examine the disk directories and file contents.

Summary

This chapter has presented the test design and procedures for the I/O Interface software development (see Table 4-I) and discussed the methods and objectives for each test. The System Integration Test validates that the I/O Interface Protocol implementation on the 432 Micromainframe computer system meets the objectives described in Chapter II; basically, to provide virtual device operation within a flexible system design. The next chapter discusses the test results and, then, presents conclusions and recommendations for future work with the AFIT/ENG 432/670 Computer System.

V. RESULTS, CONCLUSIONS AND RECOMMENDATIONS

Introduction

This investigation has been concerned with the development of an interface protocol for I/O device control in a distributed multiprocessor environment. The first three chapters have presented the design and, the fourth chapter, validation procedures for the software system. This chapter completes the written development of the I/O Interface. The following sections describe the test results, discuss the conclusions reached and present recommendations for future work with the Intel 432/670 Computer System.

Test Results

This section presents the results of the test procedures described in Chapter IV. The procedures were structured to individually test each sublayer of the I/O Interface implementation on the 432 and AP processors. The following paragraphs discuss the major results of these tests, which are summarized in Table 5-I. Detailed test results may be found in Appendix K.

Environment Validation Test. The first test of the Intel 432/670 Cross Development System was a success. The PRIME program was compiled, linked, and executed according to the procedures outlined in Chapter IV. In addition, the

TABLE 5-I

Summary of Test Results

Test	Result
Environment Validation Test	Validated
432 Processor Validation Tests:	
User Sublayer	Validated
User Agent Sublayer	NOT Validated
Message Transfer Sublayer	NOT Validated
Attached Processor Validation Tests:	
User Sublayer	Validated
User Agent Sublayer	Validated
Message Transfer Sublayer	Validated
System Integration Test	NOT Validated

PRIME program was modified to use the full implementation of the iMAX operating system (see Appendix C) and the resulting program was, also, successfully tested. Sample output listings from these tests are provided in Appendix K.

432 Processor Tests. Only the User Sublayer of the 432 processor software was validated by the test procedures of Chapter IV. The size of the executable module, containing more than the uppermost sublayer, exceeds the capacity of the double-density disk system on the Series III MDS Debugger System. The largest component of the software system is the iMAX operating system. The operating system was provided as a single module that must be linked with user programs to create an executable module for the 432

processor. The complete module must be moved to a diskette on the Debugger system before loading into the 432s memory. Since it was not possible to create a usable (small enough) module that contained more that the first sublayer of the I/O Interface, further testing of the 432 processor software was not possible.

The User Sublayer, however, demonstrated the operation of the three system commands on the 432/670 Computer System. The HELP command provides text information to the Debugger console and the SET and COPY commands responded with flag messages indicating calls to the dummy procedures of the test package. Error checking, on command line syntax, was also validated by testing.

Attached Processor Tests. The test procedures, for validating the Series III MDS AP software, were all successful. The use of a test shell program allowed validation of all functions of the I/O Interface software. The I/O devices implemented (printer, console and disk systems) all responded correctly to valid, and invalid, messages generated by the test shell program. Appendix K contains listings of the system outputs during each test.

Conclusions

The major goal of this investigation was to implement an interface system on the Intel 432 Micromainframe Computer System using a message based protocol. The concepts and

structure of the CBMS and the ISO Open System Interconnection Reference model were used to organize the system into a hierarchy of functional protocols.

The ISO Reference model provided the overall organization. The I/O Interface system was designed within this structure to a large extent. However, the model was not strictly followed at the lowest level of this design. The message management functions of the 432's operating system (iMAX) were used to manipulate the I/O Interface messages within the Message Transfer Sublayer. Strict adherence to the ISO model would have required a Transport Layer protocol providing a reliable message handling system. Implementation of the Transport layer would be useful because it would allow communications with other ISO standard systems (e.g. DELNET) and it would provide a well-known standard interface for future system users.

Future implementation of the lower levels of the ISO model are possible. The object-oriented I/O Interface design insures that the only software needing modification is the Message Transfer Sublayer. Also, the HELP command implementation was designed to provide information from fixed memory storage of the data. A more practical approach would be to use files to store the data and, then, allow the user to direct the output to any system device. This should be done using the functions of the I/O Interface. The modifications would only affect the System Commands Package

of the 432 processor software.

Modifications to the MTS and the HELP command are just two examples of the system maintainability derived from object-oriented design methodology. The placing of design features within software packages created a manageable and flexible system which may be maintained and improved in a systematic manner.

In summary, while the implementation of the I/O Interface cannot be evaluated completely until the entire system has been tested, the development of the interface as a CBMS system is a step towards a system that can be expanded to include many other computer systems and their devices through interface with a local area network organization. The realization of the DELNET system will allow continued development along these lines.

The 432 system has a powerful organization, but, it must become fully operational before a proper evaluation can be achieved. At the present time, development work with the 432 is very frustrating. The incomplete implementation of the Ada compiler and the tedious process required for software development are, perhaps, the worst features of the system.

Future users will still have to contend with the complexity of the system. Grasping the organization of the hardware and software, necessary to develop and execute programs on the 432, can be a formidable task. An effort

must be made, to reach a point where the interaction of system components is understood by the designer.

The 432 University User's Group, an organization of 432 system academic users, is aware of these problems (Ref 39). Efforts are being made to remove the Series III MDS from the development environment of the 432. When completed, the Debugger system will be an Attached Processor system executing iRMX 86 software which is common to the AP portions of iMAX. In addition, the group is developing a high-speed communications link from the VAX 11/780 system to the AP system using iRMX 86. These projects are intended to simplify the development environment and decrease the time necessary to move programs among the systems within the Cross Development System for the 432 (currently, it takes about an hour for each cycle of source modification, compile, link, download to the debugger and re-testing a program on the 432).

Recommendations for Future Study

Recommendations for further study related to this thesis effort are presented in two categories. The first group contains topics related to inter-process communications, independent of implementation. The second category is the recommended projects using the Intel 432/670 Micromainframe computer system. The projects are summarized in Table 5-II.

TABLE 5-II

Summary of Recommendations for Future Study

Network Communications Projects
1. Process Control Protocol 2. System State Information Protocol 3. Transport Layer for Delnet Interface
Intel 432/670 Computer System Projects
1. Data Flow Architecture Study 2. Data Base Processor System 3. Distributed Operating System 4. Operating System Shell for iMAX 5. Ada Language Software Test and Verification Environment 6. Other Compilers for the 432 7. Attached Processor Front-End

Inter Process Communications Projects. Several projects are possible in the area of inter-process communications in a distributed processing environment.

1. Process Control Protocol. Design, Development and possible implementation of a protocol for process control in a distributed multi-processor environment. In a distributed processing environment the kernel of the operating system must provide a method for creating, starting, stopping, and destroying processes an execution system which may not be local to the processor executing the function. Ideally this method should be uniform across the entire system. That is, the requesting process performs the same action

irrespective of the actual location of the process being operated upon. This project may begin with the assumption of an existing message based inter-process communications system.

2. System State Information Protocol. Distributed operating systems using a message based system for inter-processor communication require a method for passing state information among the systems. There is a need for a protocol defining the procedures for heterogeneous multiprocessor systems to relate state information to each other for a practical distribution of operating system control decisions.

3. Transport Layer for DELNET Interface. Implementation of a network transport layer, within the structure of the I/O Interface, would allow the 432 system to be interfaced with a local area network, such as the DELNET system. Interconnection with other computers, through a network, would allow future development of protocol systems within higher levels of the ISO Interconnection Reference Model.

Intel 432/670 Computer System Projects. Based on the continued work by Intel Corporation to improve the state of the Cross Development System for the 432 and fully implement the Ada language, the following recommendations are

presented for future work with the iAPX 432 Computer System:

1. Data Flow Architecture Study. The multi-processor organization of the 432/670 Micromainframe system presents possibilities for design studies in data flow architecture. Some of the Attached Processors might be programmed for performance of specific manipulations and the 432's GDPs could provide the flow control and processor allocation functions. Alternatively, specific GDPs might be selectively allocated to specific tasks and the Attached Processors restricted to performing the I/O (flow control) functions.

2. Data Base Processor System. The Attached Processors of the 432 System could be programmed to perform pre-processing as well as input and output functions for a data base management system using the Intel 432 Micromainframe computer system. Such an arrangement might be developed easily using the iMAX 432 executive as the system kernel. Also, various arrangements of multiple 432 GDPs could be evaluated to test different algorithms for use on multiprocessor data base management systems.

3. Distributed Operating System. The present configuration of the iMAX 432 operating system places the 432 processors at the top of a hierarchical organization. The processing power of the Attached

Processors could be used to create a more balanced system structure. This would encompass many of the considerations of the design projects listed above, related to inter-process communications.

4. Operating System Shell for iMAX. The iMAX 432 Multifunction Applications Executive for the Intel 432 Micromainframe computer system provides a functional interface to the user and is intended to be linked into the user's software as necessary. Development of a multiuser operating system shell for the 432 System using iMAX 432 as the kernel would provide the basis for a future development system. This work could be done as a follow-on project to the design of a multiprogramming operating system by Ross (Ref 34:1).

5. Ada Language Software Testing and Validation Environment. The 432 Computer System provides an execution environment for Ada language programs, within the restrictions of the limited implementation of Ada in the Cross Development System. Future releases of the compiler should provide a more complete implementation of the Ada language. However, even the existing release gives some capability for testing Ada software. As more projects are completed using this system, a software base will be built up. To ensure this is done efficiently, studies should be made to

develop configuration management techniques and perhaps even rules of form for the Ada packages to be added to this data base. This project would be primarily a software engineering effort.

6. Other Compilers for the 432. At present, only the Ada language is available to programmers of the Intel 432 Computer System. Writing compilers for other languages in Ada, for execution on the 432 system, would allow software to be transported from other systems and re-compiled for execution on the 432.

7. Attached Processor Front-End. The attached processor is the I/O processor for the 432 computer system. Increasing the intelligence of this I/O processor will allow the main system (the 432 processors) to perform their remaining tasks more efficiently. One possible improvement would be to move a portion of the operating system shell intelligence to the attached processor. For example, while acting as the terminal handler and file storage system monitor for the main system, the attached processor could also intercept requests for text file displays and provide the requested file data without communicating with the 432 GDPs. Such an arrangement would constitute a small step towards a distributed operating system and could be used to evaluate the potential for more cooperative

processing among the components of the 432 System.

As indicated by the range of these projects, there are a large number of uses for the 432 Computer System that are unexplored. In addition, the 432 University User's Group (Ref 39) can provide a source of information on the current directions for research with the 432.

Bibliography

1. ANSI/MIL-STD-1815A. Ada Programming Language. Washington, D.C.: United States Government, Under Secretary of Defense, Research and Engineering, 1980.

2. Coffman, Edward G. Jr. and Peter J. Denning. Operating Systems Theory. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1973.

3. DARPA Publication RFC: 791. Internet Protocol: DARPA Internet Program Protocol Specification, DOD Standard. Arlington, Virginia: Defense Advanced Research Projects Agency, Department of Defense, 1981.

4. Hansen, Per Brinch. Operating System Principles. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1973.

5. Hemenway, Jack. "Object-Oriented Design Manages Software Complexity," EDN, 16:141-145 (August 1981).

6. Hoare, C. A. R., and R. H. Perrott. Operating Systems Techniques: Proceedings of a Seminar held at Queen's University, Belfast, 1971. London, United Kingdom: Academic Press Inc., 1972.

7. Holmes, Victor P., Bruce N. Malm, and Tom H. Little. "Island Universes: Distributing a Single-User Operating System," Proceedings of the Conference on Parallel Processing, 1982, 319-321. New York: Institute of Electrical and Electronics Engineers, Inc., 1982.

8. Institute for Computer Sciences and Technology pubilcation CBOS-82-1. Features of a Message Transfer Protocol, Draft Report. Gaithersburg, Maryland: Institute for Computer Sciences and Technology, National Bureau of Standards, November, 1981.

9. Institute for Computer Sciences and Technology publication CBOS-82-4. Naming and Addressing in Computer Based Message Systems, Draft Report. Gaithersburg, Maryland: Institute for Computer Sciences and Technology, National Bureau of Standards, August 1982.

10. Institute for Computer Sciences and Technology publication CBOS-82-3. Service Specification of a Message Transfer Protocol, Draft Report. Gaithersburg, Maryland: Institute for Computer Sciences and Technology, National Bureau of Standards, February 1982.

11. Institute for Computer Sciences and Technology publication LANP-80-2. Standards for Local Computer Networks, Draft Report. Gaithersburg, Maryland: Institute for Computer Sciences and Technology, National Bureau of Standards, March, 1980.

12. Institute for Computer Sciences and Technology Proposed Federal Information Processing Standard Specification for Message Format for Computer Based Message Systems. Gaithersburg, Maryland: Institute for Computer Sciences and Technology, National Bureau of Standards, April 1982.

13. Intel Publication No. 121618-003. Intellec Series III Microcomputer Development System Programmer's Reference Manual. Santa Clara, California: Intel, Corp., 1981.

14. Intel Publication No. 142603-004. iRMX 80/88 Interactive Configuration Utility User's Guide. Santa Clara, California: Intel Corp., 1981.

15. Intel Publication No. 143232-002. iRMX 88 Reference Manual. Santa Clara, California: Intel Corp., 1981.

16. Intel Publication No. 143241-003. iRMX 88 Installation Instructions. Santa Clara, California: Intel Corp., 1981.

17. Intel Publication No. 171858-001 Rev. B. iAPX 432 Object Primer. Santa Clara, California: Intel Corp., 1981.

18. Intel Publication No. 171821-001. Introduction to the iAPX 432 Architecture. Santa Clara, California: Intel Corp., 1981.

19. Intel Publication No. 171867-001. Intel 432 System Summary: Manager's Perspective. Santa Clara, California: Intel Corp., 1981.

20. Intel Publication No. 171869-002. Reference Manual for the Ada Programming Language. Santa Clara, California: Intel Corp., 1981.

21. Intel Publication No. 171870-002. Intel 432 Cross Development System VAX Host User's Guide. Santa Clara, California: Intel Corp., 1982.

22. Intel Publication No. 171954-002. Introduction to the Intel 432 Cross Development System. Santa Clara, California: Intel Corp., 1982.

23. Intel Publication No. 172097-002. Intel 432 Cross Development System Workstation User's Guide. Santa Clara, California: Intel Corp., 1982.

24. Intel Publication No. 172098-002. System 432/600 System Reference Manual. Santa Clara, California: Intel Corp., 1982.

25. Intel Publication No. 172103-002. iMAX 432 Reference Manual. Santa Clara, California: Intel Corp., 1982.

26. Intel Publication No. 172174-001. Asynchronous Communication Link User's Guide. Santa Clara, California: Intel Corp., 1981.

27. Intel Publication No. 172283-001. Reference Manual for the Intel 432 Extensions to Ada. Santa Clara, California: Intel Corp., 1981.

28. Kahn, Kevin C. and Fred Pollack. "An Extensible Operating System for the Intel 432," Proceedings of the Twenty-Second Computer Society International Conference, 398-404. New York: Institute of Electrical and Electronics Engineers, Inc., February 1981.

29. Kahn, Kevin C., William M. Corwin, T. Don Dennis, Herman D'Hooge, David E. Hubka, Linda A. Hutchins, John T. Montague, and Fred J. Pollack. "iMAX: A Multiprocessor Operating System for an Object-Based Computer, "Proceedings of the Eighth Symposium on Operating Systems Principles, 15 (5):127-136, Association for Computing Machinery, December 1981.

30. McNamara, John E. Technical Aspects of Data Communication. Bedford, Massachusetts: Digital Equipment Corporation, 1977.

31. Moulton, James. "High Level Protocol Boundaries in the ISO Model," IEEE 1980 Trends and Applications: Computer Network Protocols, 54-58. New York: Institute of Electrical and Electronics Engineers, Inc., 1980.

32. Phister, Paul W., Jr. Protocol Standards and Implementation within the Digital Engineering Laboratory Computer Network (DELNET) using the Universal Network Interface Device (UNID). Unpublished MS thesis. Wright-Patterson AFB, Ohio: School of Engineering, Air Force Institute of Technology, October 1983.

33. Rattner, Justin and George Cox. "Object-Based Computer Architecture," Computer Architecture News, 8 (6):4-11 (October 1980).

34. Ross, Mitchell S. Design and Development of a Multiprogramming Operating System for Sixteen Bit Microprocessors. MS thesis. Wright-Patterson AFB, Ohio: School of Engineering, Air Force Institute of Technology, December 1981.

35. Smith, Lynn M. Intel 432/670 Computer System User's Guide. Unpublished text. Wright-Patterson AFB, Ohio: School of Engineering, Air Force Institute of Technology, June 1983.

36. Smith, Lynn M. Investigation of the Interfacing of the Intel 432/670 Computer System to the MIL-STD-1553B Serial Avionics Bus. Unpublished MS thesis. Wright-Patterson AFB, Ohio: School of Engineering, Air Force Institute of Technology, June 1983.

37. Stankovic, John A., Andries van Dam. "Research Directions in (Cooperative) Distributed Processing," Research Directions in Software Technology, edited by Peter Wegner. Cambridge, Mass.: The MIT Press, 1979.

38. Weinberg, Victor. Structured Analysis. New York, New York: Yourdon Press, 1979.

39. Weaver, Alfred C., Professor. 432 University User's Group correspondence. Department of Computer Science, Thorton Hall, University of Virginia, Charlottesville, Va., October 5, 1983.

40. Zeigler, Stephen, Nicole Allegre, Robert Johnson, and James Morris. "Ada for the Intel 432 Microcomputer," Computer, 18:47-56 (June 1981).

41. Zeigler, Stephen, Nicole Allegre, David Coar, Robert Johnson, and James Morris. "The Intel 432 Ada Programming Environment," Proceedings of the Twenty-Second Computer Society International Conference, 405-410. New York: Institute of Electrical and Electronics Engineers, Inc., February 1981.

APPENDIX A

Intel 432/670 System Architecture

Introduction

The purpose of this appendix is to provide an overview of the hardware and software architectures of the Intel 432/670 Micromainframe Computer System. This introduction presents the major features of the iAPX 432 architecture. The remaining sections describe the memory organization, software structures, and component architectures of the system.

Intel claims that "the iAPX 432 represents one of the most significant advances in computer architectures since the 1950s" (Ref 18:1-5). The list of features presented to justify that statement includes the following (Ref 18:1-5/1-7):

-- The iAPX 432 is the first computer architecture designed to support software-transparent, multi-processor operation.

-- The iAPX 432 is the first commercial computer whose architecture fully supports the new object-oriented programming design methodology.

-- The iAPX 432 is designed to be programmed entirely in high-level languages.

-- The iAPX 432 has a large virtual address space (2**40 bytes) and hardware mechanisms for implementing

virtual memory systems that can use this environment.

-- The iAPX 432 hardware can handle fault conditions in a multiprocess, multiprocessor environment and allow other system processes and processors to continue running.

The architecture of a 432 "system" is also described by its components. The system consists of an interconnected system of components, which include: one or more General Data Processors (GDPs), a Main Memory System, and at least one Interface Processor (IP) connected to an I/O system. Figure A-1 shows a minimum system configuration.

In order to obtain high performance in both general processing and input/output operations, the iAPX 432 has a distinct type of processor for each function. The GDP handles all program decoding, computation, and address generation. The IP performs all communication with peripheral devices. Communication among the GDP, IP and memory is provided by a packed-based interconnect bus. The IP is also connected to an interrupt-driven I/O system bus. The I/O system also contains the I/O devices and a conventional processor, called the Attached Processor (AP).

These components are discussed in following sections of this appendix. More information is also available in the Intel Introduction to the iAPX 432 Architecture (Ref 18). The memory organization, which is presented first, provides a basis for the software structures and component architectures described later.

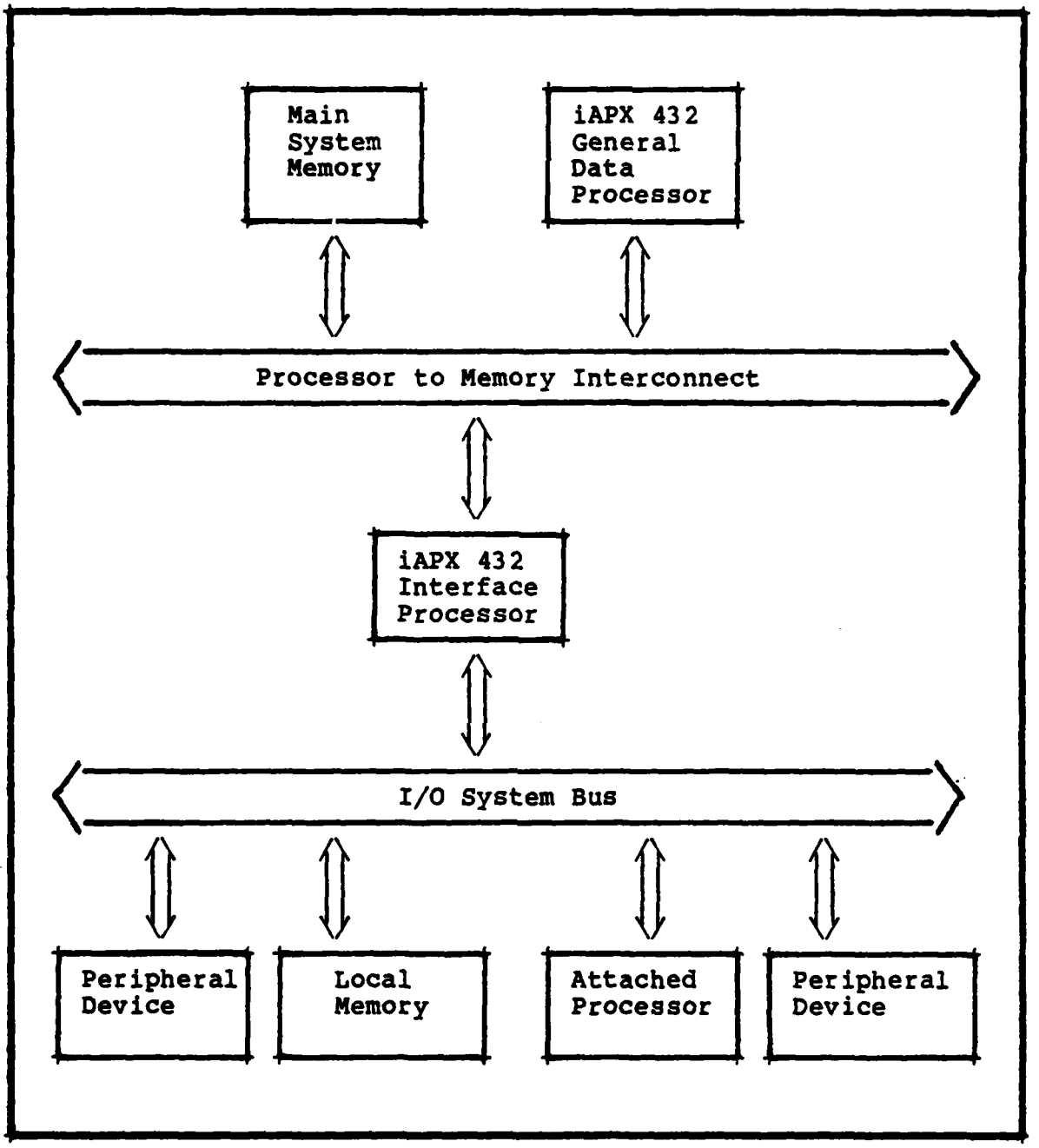


Figure A-1. iAPX 432 Minimum System Organization

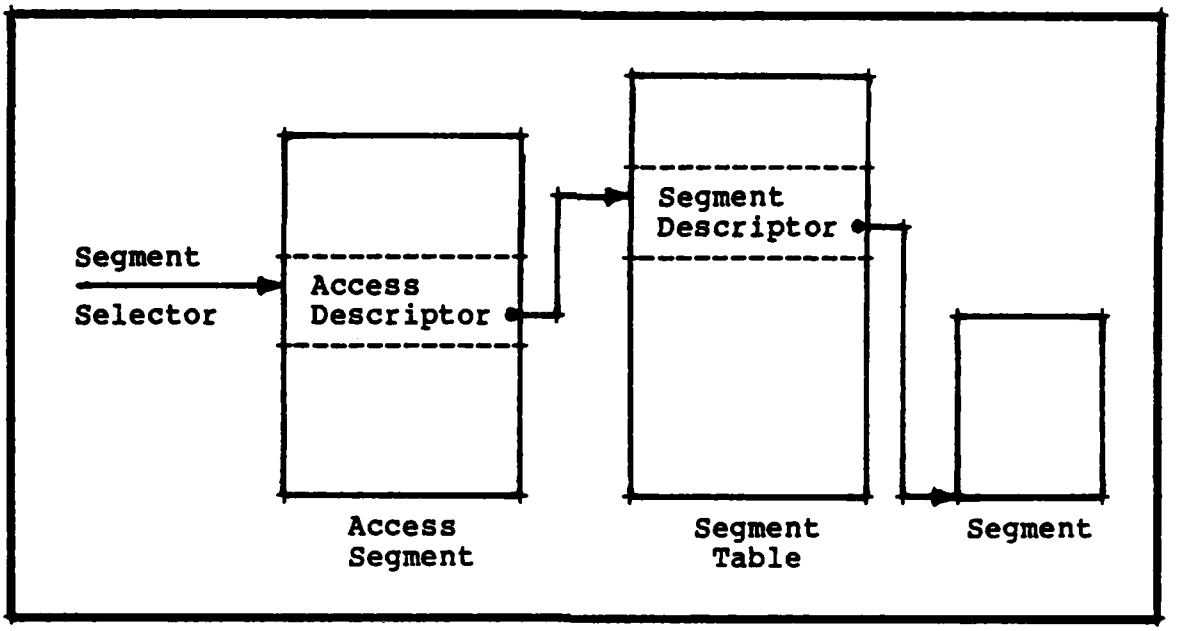


Figure A-2. Two-Level Mapping (Ref 18:2-11)

Memory Organization

The iAPX 432 system uses segmented memory. However, Intel uses the term "structured memory" to refer to the organization because it is enhanced in two ways. First, the iAPX 432 can address 2**24 memory segments, which may each specify up to 64K bytes of memory. Thus, the virtual address space is 2**40 bytes. Second, the 432 uses an additional step in the address mapping process which is unique to the iAPX 432. This two-stage mapping separates segment relocation from access control and allows implementation of segment type checking and user-access rights in hardware (Ref 18:2-10).

In a 432 system, each program module is supplied, at run-time, with a collection of segment numbers (i.e., values

that can be used as indices into the segment table to select segment descriptors) for only the segments that it may need to access during execution. This collection is stored in a set of access segments (tables of access descriptors). This is illustrated in Figure A-2.

Each logical address consists of two 16-bit components, the segment selector and the displacement. The segment selector is an index into the proper access segment. The access descriptor contains access rights information which is checked by the domain-protection mechanism during each memory access, and an index to the segment descriptor describing the memory segment needed. The segment descriptor contains the 24-bit base address of the segment and the length of the logical memory object. The base address is added to the displacement, giving the physical address of the operand within the segment. The base address and the displacement create the virtual address space of 2**40 bytes (Ref 25:KEY-1/KEY-4). All addressing within the main system memory is performed with this mechanism. In particular, each reference to a data structure in main memory, by the GDP or IP, is done using the two-stage addressing. The software structures of the GDP and IP programs are described in the next section.

Software Structures

432 system environments, both hardware and software, have been developed using the object-oriented design

methodology. Many of the facilities implemented in the 432 hardware architecture are best utilized by programs designed using this same methodology. For example, the access-checking and protection mechanisms are used to enforce the execution environment of the program module. As explained in Appendix B, object-oriented design methods lead to data structures whose use is confined to a single program module. That is, all the facilities which physically manipulate a particular data structure are contained in a single module and thus, no other program module has a need to directly access that data structure.

In programming methods which are structured by function, several modules might contain procedures which would need information from the data structure and so, each code segment or program module would require access to the data structure. While the 432 access-checking and protection facilities would support such an arrangement, there is less protection provided when all the program modules have legitimate rights to access a data structure than when the software design specifically limits the need for direct access to the data structures.

Software designed with the object-oriented methodology tends to hide detailed design decisions within the program modules (Ref 5:143). The logical view of the data structure, or object, is that of a collection of functions which provide for manipulation of the data without knowledge

of its precise structure. Thus, each software module is a logical, or virtual, machine providing necessary functions to other machines.

This "virtual machine" concept extends to the larger view of a complete software system. The basic goal of an operating system is to provide an easily-used superset of the actual microprocessor hardware functions (Ref 38:1-3). Object-oriented software may be built up easily with higher level modules using the functions provided by other modules to implement more complex system structures. Intel's design engineers used this methodology in the construction of the Multifunction Applications Executive for the 432 Micromainframe (iMAX). The iMAX software is provided as a set object modules. Appendix C summarizes the features of the iMAX operating system.

General Data Processor Software Structure. In the 432 GDP, the hardware-software interface occurs at a higher level than in conventional systems (Ref 19:15-26). In conventional microprocessor architectures, the major facilities include processor registers, various addressing modes, and built-in data types with their associated operations. Very large scale integration (VLSI) technology is used in the 432 architecture to implement functions which are normally found in operating system software. The introduction of system objects as hardware recognized entities is a fundamental step toward a higher level of

interface. Essentially, this means the processor is not limited to the simple data types normally recognized by a microprocessor architecture. This is further explained in the Intel iAPX 432 Object Primer (Ref 17:2-5). The facilities provided by the 432 architecture include the following features:

-- Access checking and protection mechanisms for enforcing program modularity.

-- Execution environment for program modules.

-- Scheduling and communication mechanisms for multiple software tasks.

-- Control and dispatching for multiple hardware processors.

Thus, the system designer is not burdened with creating an environment of basic operations to manipulate the system elements; tasks, program modules, and data structures. The facilities to handle these objects are provided by the hardware.

The task communication facilities provided by the hardware are supported by the operating system software and provide for the management of port and message objects. The operating system provides for communications between hardware processors by allowing different processors to perform the sending and receiving tasks on the same communications port object. This provides for a consistent design environment within a multiprocessor system.

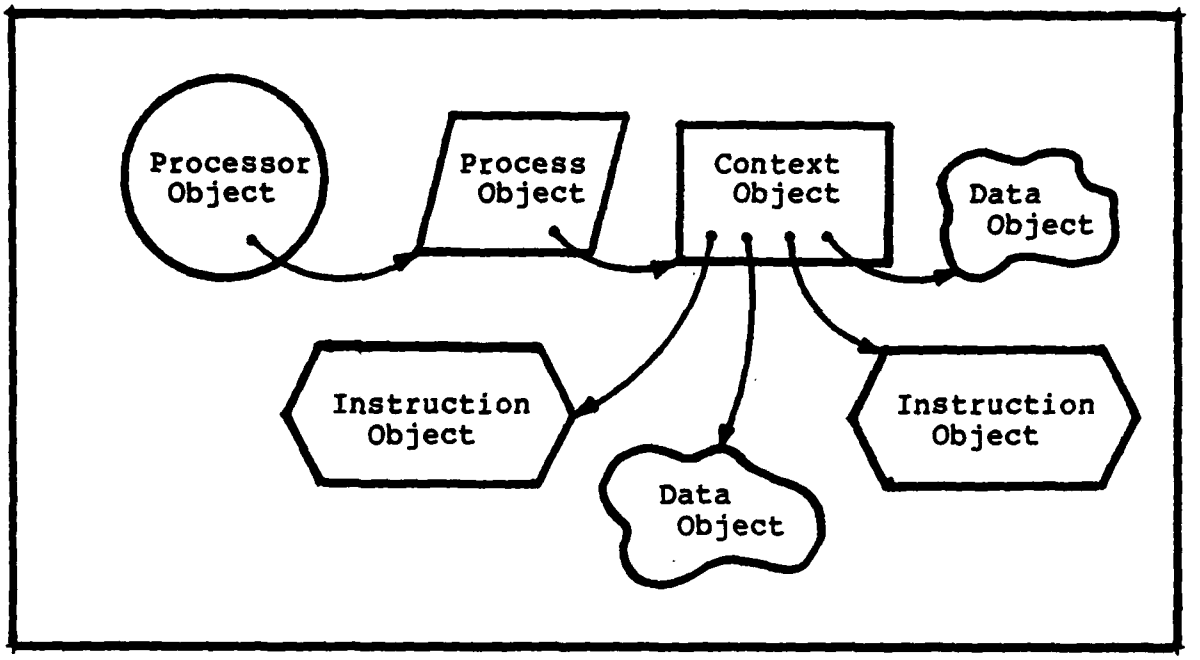


Figure A-3. GDP Program Structure

A program which runs on the 432 GDP consists of five types of objects; processor objects, process objects, context objects, instruction objects, and data objects. The basic GDP program structure is shown in Figure A-3.

Each physical processor has its own processor object which contains state information (*running*, halted, etc.), diagnostics, and references for system objects. The object references are important because the addressing and protection mechanisms only allow a processor to access the memory for which it has object references. One of the system objects that is referenced by the processor object is the process object of the currently executing process.

There is at least one process object for every program

and exactly one process object for every process in the system. The process object contains the state information for the process, scheduling information, and a reference for the context object currently being executed. Each process, in a program, consists of one, or more, procedures.

A procedure is a set of instructions logically grouped together to perform a specific operation. Each procedure is represented by the third program object type, the context object. Since many processes may call the same procedure, a copy, or instance, of the variable parts of the procedure is created for each reference by a process. Each instance of a procedure is given its own context object which contains references to the instruction and data objects used by the procedure. This set of references is called the access environment of the procedure. The protection mechanism of the iAPX 432 does not allow a procedure to access any area of memory without the proper reference in its access environment (Ref 17:3-5/3-17).

All of the necessary instructions and data for a procedure are stored in instruction and data objects. The instruction objects contain only instructions and the data objects contain only data. This separation improves the system reliability because the protection mechanism can ensure that only instructions are executed. Also, all programs are reentrant since the instruction objects may not be modified during execution.

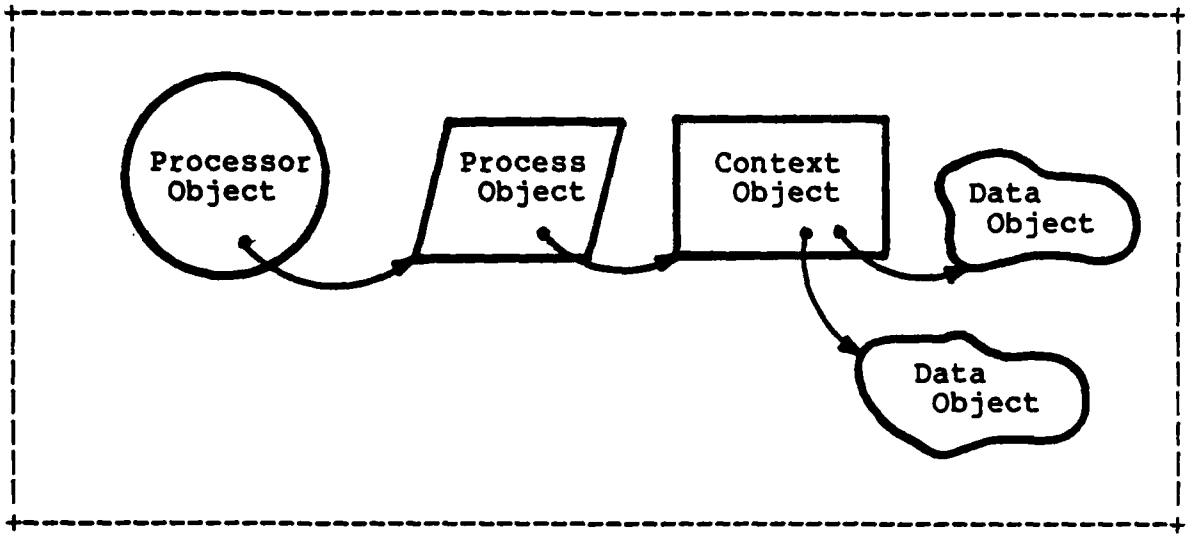


Figure A-4. IP Program Structure

The data objects are the fifth type of object in a GDP program. They contain only data. The type of data they contain is defined by the software. Data objects are the only GDP program objects that cannot be directly manipulated by the hardware (Ref 17:3-18/3-19).

Interface Processor Software Structure. The IP program structure is similar to that of the GDP (see Figure A-4). The major differences result from the IP having no capability to fetch its own instructions. The IP is a slave to the Attached Processor (AP). Basically, the IP system objects contain the same information as their counterparts in the GDP system. However, the context objects do not contain references to the IP instructions. Instead, the IP possesses a function request area, where the AP can write instructions to be executed by the IP, and a status

information area, which the AP can read to determine the result of the last instruction.

IP objects are, also, protected in the same manner as the GDP objects. However, the protection mechanism can be turned off by the AP. This feature is used, during system initialization and diagnostic support, to permit the AP to more directly access the memory of the 432 system and is called physical reference mode. Normal interprocessor communication is done in logical reference mode, where the access protections of the GDP are extended to the AP when it is accessing the 432 system main memory through the facilities of the IP.

The iAPX 432's program architecture is very complex. Greater detail is provided in the appropriate Intel reference manuals and user's guides for further understanding.

Hardware Architecture

The Intel iAPX 432 Micromainframe hardware is designed for multi-processor applications where small physical size, low power consumption and dependability are essential (Ref 8:1). Intel's engineers used a new approach to computer technology when designing the VLSI components of the iAPX 432 family. They integrated the hardware and software design methodologies while aiming to reduce the life-cycle costs of complex microcomputer applications (Ref 19:2). The result is a computer system incorporating innovations which

constitute a new computer technology (Ref 19:18).

To accomplish a significant improvement in life-cycle costs, the designers of the 432 micromainframe incorporated the software design into the hardware architecture. The current trend in lower cost hardware and higher cost software led to considerations for simplifying the software design task and reviewing the facilities necessary to software development, testing, and maintenance. Toward this goal, object-oriented design methods were used while incorporating operating system functions into the hardware design (An explaination of object-oriented design is presented in Appendix B).

Hardware access-checking and protection mechanisms are also implemented into the 432's VLSI design, which means that there are software errors that are impossible on the 432 (Ref 17:1-2):

-- A module can not access data beyond the range of its own environment.

-- A module cannot modify a data structure that it should only read, nor read where it should only write.

-- A module cannot perform an operation that is not allowed for the type of data it is using. (e.g., It cannot execute data for instructions).

The individual components of the iAPX 432 family have all been designed with unique features to maximize the system performance and provide the system designer with a practical

environment for the development of complex systems.

General Data Processor Architecture. The GDP is responsible for the computational operations of the iAPX 432 system. The architecture of the GDP does not support interrupts, service requests, or possess user-visible registers, as do conventional microprocessors. Instead, the GDP is logically organized as a three-stage microprogram-controlled pipeline. The iAPX 43201 Instruction Decoding Unit contains the Instruction Decoder and the Microinstruction Sequencer. The iAPX 43202 Microinstruction Execution Unit contains the last stage of the pipeline, the Execution Unit. Each stage operates independently. This pipeline organization is designed to efficiently implement the complex instruction set of the iAPX 432 (Ref 18:3-1/3-9).

The instruction coding scheme of the iAPX 432 is designed to minimize the amount of memory necessary for the instructions while allowing for efficient decoding. The instruction format consists of an ordered set of variable length fields; class, format, reference, and opcode. The class field specifies the number of operands and the data type of each. The format field is used to specify the mapping of the required operands to the operand references. The format field is only included when the class field indicates that operands are included in the instruction. The reference field contains the explicit operand references

(when indicated by the format field). The instruction set is completely symetric, allowing any, of four, addressing modes with each operand. The last field of the instruction is the opcode field which specifies the operation to be performed (Ref 18:3-5/3-7). Details of the instructions and addressing modes may be found in the Intel Introduction to the iAPX 432 Architecture (Ref 18) and the Intel iAPX 432 General Data Processor Architecture Reference Manual.

Interface Processor Architecture. The iAPX 43203 Interface Processor performs the communication functions between the main system memory and the AP system. The 43203 processor chip contains two independent functional units; the Data Acquisition Unit, which transfers data between the AP's system memory and the 432's main memory, and the Microinstruction Execution Unit, which executes the IP instructions. The AP controls the operations of both these units. The AP requests service from the IP by writing the opcode and operands, of a 432 IP macro-operator, into the IP's control object which is stored in the 432's main memory. The status of the operation may be read by the AP to determine if the desired function has been successfully completed. Details of the iAPX 432 IP chip's architecture and operation can be found in the Intel iAPX 432 Interface Processor Architecture Reference Manual and the Intel iAPX 432/670 Computer System User's Guide (Ref 35).

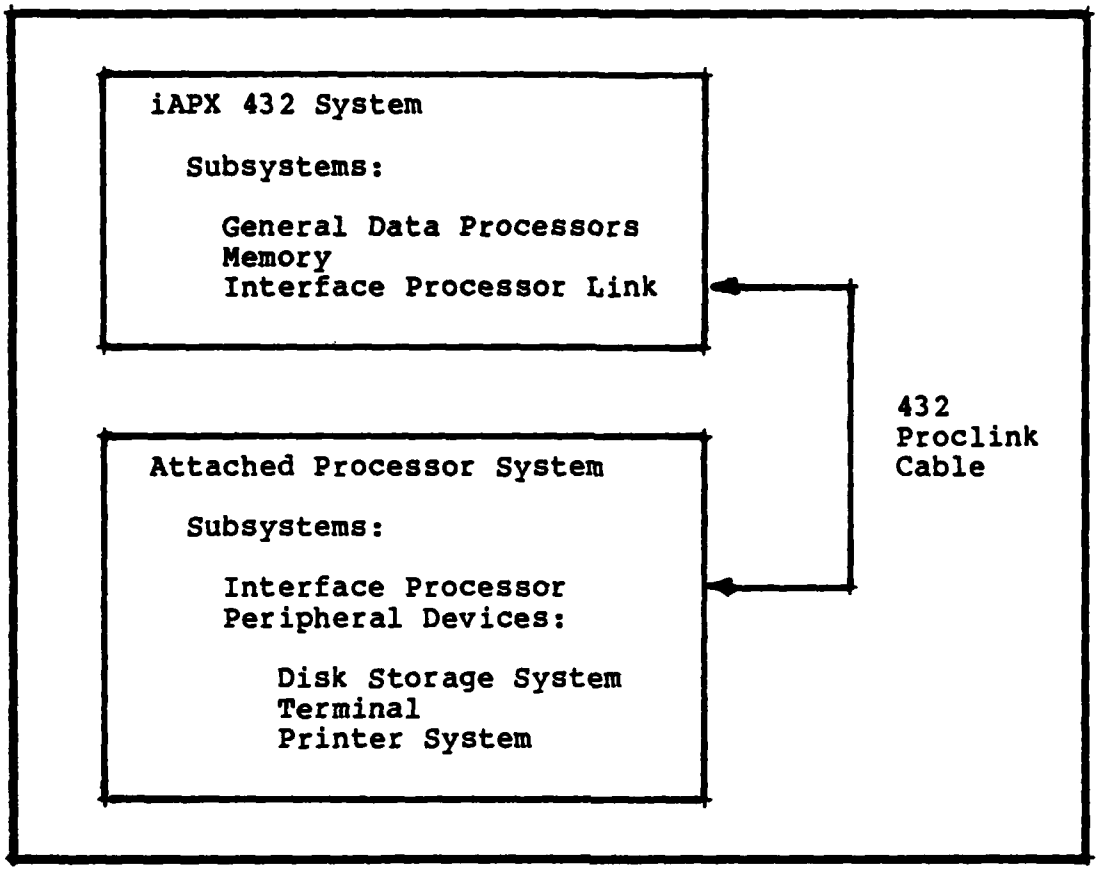


Figure A-5. General 432/670 System Configuration

<u>432/670 Architecture</u>. The 432/670 system is composed of a processor subsystem, a memory subsystem, and at least one peripheral subsystem. Figure A-5 shows this configuration.

The Interface Processor (IP) board handles all of the input/output (I/O) operations, linking the 432/670 system to external Attached Processor (AP) systems. Each AP system contains its own processor and memory and runs under a separate operating system (Ref 28:1). A 432 Proclink Cable, connecting an Interface Processor Link (IPL) board with an

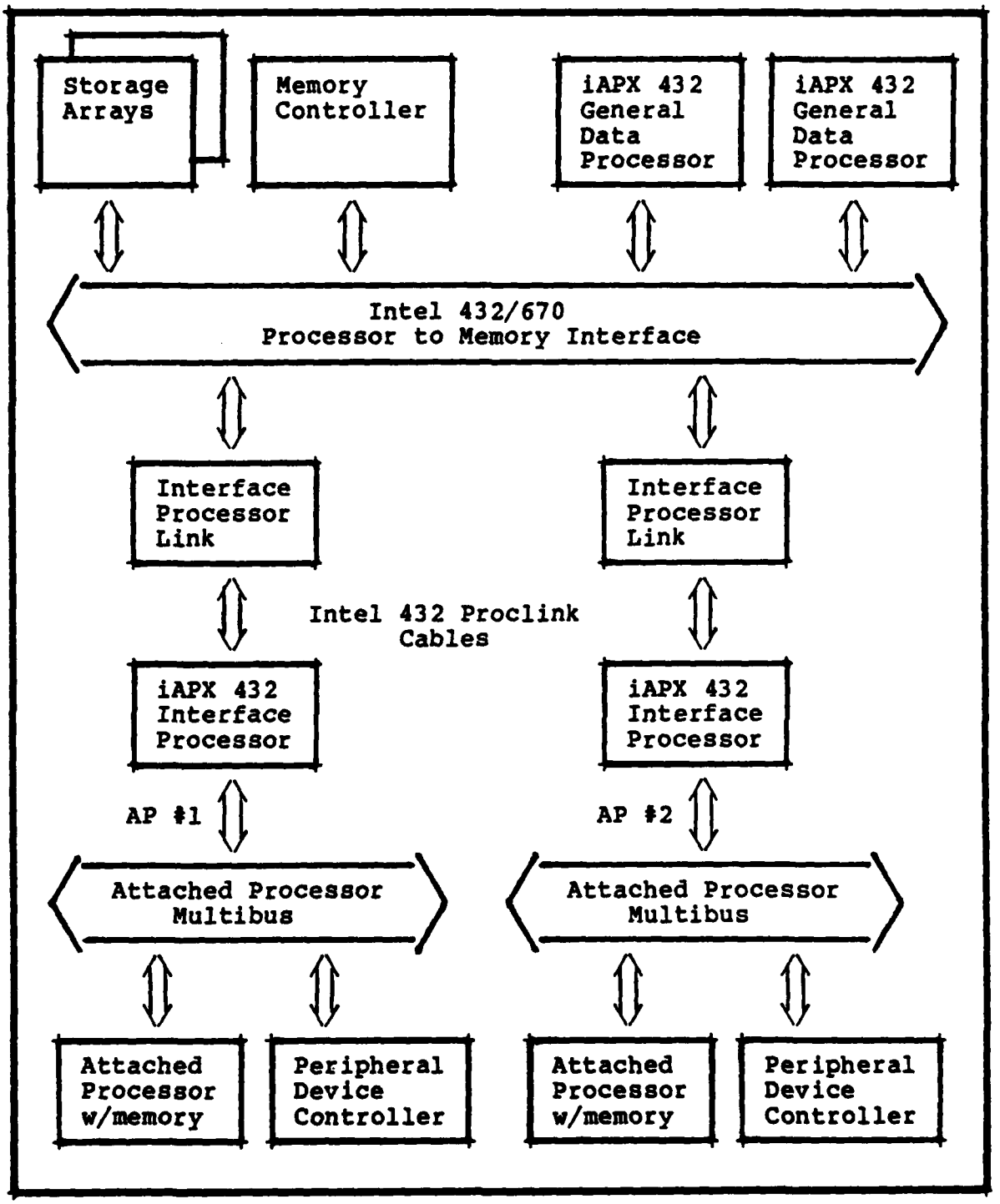


Figure A-6. Intel 432/670 Micromainframe Multiprocessor Configuration

IP board, provides the hardware connection to the main system. Figure A-6 shows the interconnections within a multiprocessor configuration of the 432/670.

The IP board must be placed on the system bus of AP the system. This limits the systems appropriate for APs to Intel MULTIBUS configured systems. Implementation of an interface with the 432 computer system involves software development on two distinct processor systems; the 432 processor system and the Attached Processor system. The 432 must be programmed to communicate with a particular AP system through the IP connected to that system. The iMAX 432 Operating System (see Appendix C) provides the basic functions necessary for this interface.

Summary

This appendix has presented an overview of the iAPX 432 system architecture. The 432 system memory was described to provide a virtual address space of 2**40 bytes over a physical address space of 2**24 bytes. The GDP and IP software structures (processor, process, context, instruction and data objects) were discussed. The architectures of the GDP and IP components were briefly described and, finally, the organization of the 432/670 system was described. If more information is required, the reader should refer to the Intel Introduction to the iAPX 432 Architecture (Ref 18) and other Intel publications.

APPENDIX B

Object-Oriented Systems Design

Introduction

Object-orientation in computer program design was first proposed by D.L. Parnas in 1972 (Ref 5:141). His technique offers a step beyond structured programming in the effort to create maintainable software that can be changed and expanded to meet future needs. Object-oriented design provides a system for developing effective data structures as well as program-control flow. This technique can provide a consistent design approach to the system architecture, operating system, and programming language.

This appendix will examine object-oriented design and its use in the Intel iAPX 432 microcomputer system, which is the first microcomputer to fully support its use in hardware and software (Ref 5:141-142). The fundamental concepts of object-oriented design will be presented and then, using the 432 system as an example, the application of the technique will be discussed in the 432's architecture and operating system design.

Object-Oriented Design

The object-oriented methodology is most unique in its consistent approach to program flow control and data structures. An "object" is a collection of procedures which encapsulate a data structure. It is the characteristics

(procedural interface) of the the object, rather than their implementation, that are of primary concern to the programmer. To demonstrate the important features of the method it is best to compare it to the more common design approach.

In conventional design methods each procedure or process becomes a module. Processes which need to use the information held in a particular data item will need to know the structure of that data. For example, if a data item contains a set of student names and test scores associated with each student, then a procedure which will get the scores for "John Doe" must have detailed knowledge of the organization of the information. The nature of the data structure is important for proper retrieval of the information (e.g., whether the names and scores are stored as records of a tape file or items of distinct arrays to be indexed by an integer). Generally, any other procedures which act upon the data must also know the structure of the data item.

With the conventional approach, functional cohesion is the goal for a system module (Ref 38:192-195). The function performed is defined by the collection of procedures contained in the module. Each well-structured module performs a single function. The interface to all other modules is completely defined by the input and output requirements. However, the connection between a function

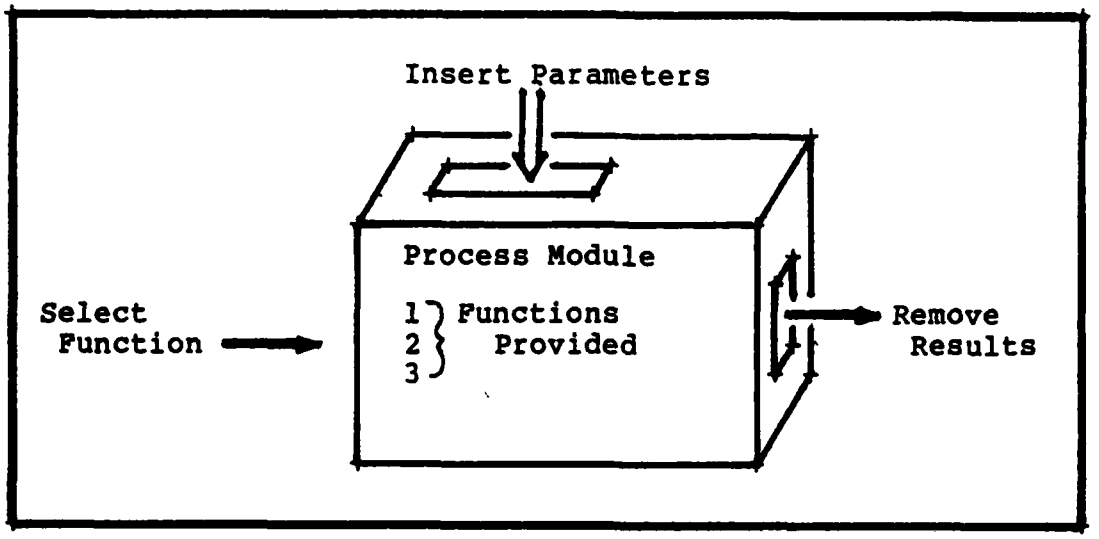


Figure B-1. A Module in the Object-Oriented Methodology

and the data items it uses must be of a global nature, that is, understood at the lowest level without explicitly having been provided as an input to the procedure.

The object-oriented approach does not require this global structure information. Object-oriented modules are built from procedures which provide functions or information to other modules in the system. The hiding of design decisions is the criteria for modularization (Ref 5:143). The exact structure of the data item would be hidden from procedures outside the object module. Outside procedures would access the information in the data item by using the functions provided by the module. Figure B-1 shows this concept as a black box which can only perform certain functions. Those functions may present the appearance of logically well-defined object but the details of the items

in the box and the workings of the functions provided are hidden from view.

This hiding feature for data items is an extension of the conventional modularized procedures which hide the complexities of program control flow in the structured program module having a single input and a single output. In general, an object-oriented module would hide program control flow ant the structure of the objects it manipulates whether they are procedures or data items.

This provides a "simple" interface between modules. There are no complex data structures which must be implicitly known within a central module, only the procedures which clearly define the characteristics of the object represented by the module. Also, the access to data items is protected by limiting the use of the data item to the procedures of object module. Thus, it is easier to understand a module's function and thereby maintain its usefulness.

Object-Orientation in Architecture

An object-oriented computer architecture would hide the physical structure of the machine and provide functions to manipulate logical storage containers while not requiring global knowledge of the hardware configuration. The Intel 432 microcomputer architecture was designed to this goal (Ref 29:125). It exhibits several important traits and

powerful features due to this design concept.

The memory of the 432 microcomputer is accessible only as a collection of objects and not as a single contiguous segment of storage. This provides for modular structure in memory use and a protection mechanism based on data structures (Ref 33:5). Unfortunately, the processing time used to perform these functions is significant. The real time, however, is minimized by implementating the procedures in VLSI hardware.

The objects in memory are classified by their use and structure (Ref 40:406). Instruction objects are clearly identified from data objects. This enables the system to know when it is accessing data or procedure code and thereby protect itself from incorrect accesses. In addition, there are two forms of objects for instructions and data; "access objects" and "simple objects". Access objects contain only access descriptors which hold the information concerning the availability of other objects to the current process and the structure of the information in those objects. Simple objects hold only data (instructions or program data) and are at the bottom of the memory structure hierarchy.

The hardware of the 432 microcomputer is constructed to present a modular view to system designers. The functions provided by the architecture raise the level of the hardware-to-software interface to logical object transformations rather than bit manipulations (Ref 33:4).

The penalty for this capability is increased hardware complexity and functional overhead as the operations on the bits must be performed at a lower level which is not accessable to the system's programmer.

Object-Orientation in Operating System Design

The use of object-oriented structure in a computer operating system can provide an environment for program development which is largely independent of the hardware implementation upon which the system operates. Modules which hide the detailed implementation of function and data structures are necessary in providing useful procedures to architecture independent systems. The Intel iMAX 432, Multifunctional Applications Executive, provides basic system functions as Ada "packages," the Ada language equivalent of objects (Ref 39:50-51).

The iMAX system provides modular functions to perform input/output, process control, and memory management. Each function is presented for use as a part of an operating system. All the necessary interface data is provided in the Ada package specification which is provided for the designer's use. However, the details of the function or the data it uses are not available to the user. This is an extreme example of device independence. The user is unable to directly access a hardware device without using a pre-defined function made available by the iMAX package which manages that device. This simplification may seem ideal to

the programmer who is willing to use the device as it is available through the operating system. However, the designer who wants to have intimate control over the device will be frustrated by the system's overhead.

In the case of memory management and protection, the object-oriented procedures of iMAX simplify the design process. The user may select from externally identical object modules which provide the same functions to the system but are internally different. For example, one system package may provide for page-swapping memory while another uses a non-dynamic approach. Whichever object the user chooses, the functions available and their inputs and outputs are the same. The user need only select the implementation which is most efficient for his program (Ref 29:132-133). The modularity of the system will allow the decision to be modified at a later date and another implementation of the same function could be used.

The process control for the system is also modular. In the Intel iAPX 432 system process control at the architecture level is defined by object modules so that a process execution is independent of the number of hardware processors in the system (Ref 28:402). A program that executes with three processors in parallel will run, without modification, on the same system with only two processors. The portion of the operating system that selects the next available process to be run can also be modified to change

its selection criteria without requiring modifications to the modules which use the package.

Conclusion

The object-oriented design technique provides a consistent approach to system organization. The same modular approach can apply to the hardware architecture as well as the system software. The object module concept provides clear function definition while hiding the design decisions made and implemented at lower levels. This modular interface to lower levels provides flexibility in design and maintainablity in the system by simplifing the interconnections among the parts of the system.

The cost of these features is in the complexity of the hardware itself and the overhead associated with manipulation with more complex data structures.

APPENDIX C

iMAX 432 Multifunction Applications Executive

Introduction

The iMAX 432 Multifunction Applications Executive (iMAX) provides executive services to user-supplied software that calls iMAX procedures. The purpose of this appendix is to summarize the features of iMAX and describe how the functions may be used in the creation of a software system for the Intel 432/670 Micromainframe computer system (432).

The primary source for the information presented in this appendix is the iMAX 432 Reference Manual (Ref 25). For a detailed description of the features of iMAX, the reader should refer to that document. Terminology used in this appendix matches that in the reference manual to provide continuity for the reader. The following sections describe the features of iMAX in five areas: Process Management, Storage Management, Interprocess Communication, Input/Output, and System Configuration.

Process Management

iMAX recognizes two types of processes; static processes and dynamic processes. Static processes are created at system initialization and are relatively inflexible. Dynamic processes are created during system execution and may be manipulated through use of the functions provided in the Basic_Process_Management package

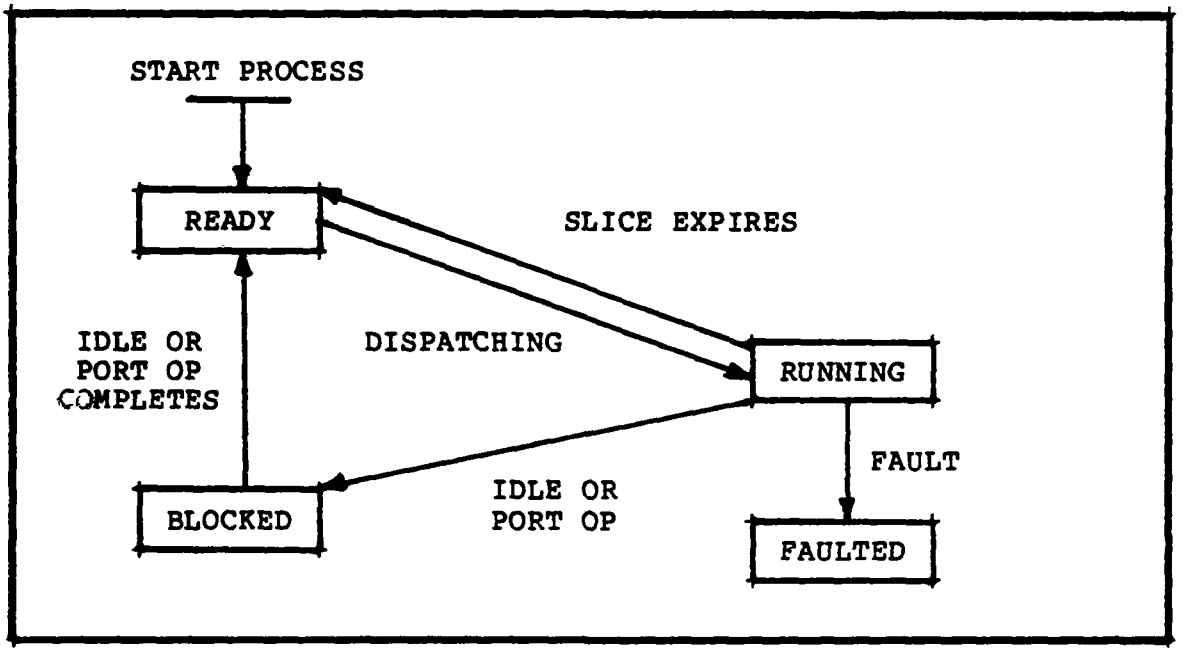


Figure C-1. Static Process State Transitions
(Ref 25:CON-3)

(BPM) (Ref 25:BPM-1). This section discusses the functions available for control of each type of process.

Static processes are processes which are defined at compile time and started at system initialization. iMAX does not provide any control operations on static processes within the Basic_Process_Management package. The only operations available on static processes are the iMAX_Definitions.Idle procedure, which causes the process to be suppended from scheduling for a given time period, and the port operations, which are described later in this appendix. Figure C-1 shows the state transitions for static processes (Ref 25:CON-3, INI-7).

Dynamic process management functions are available in

TABLE C-I

Comparison of Ada Tasks and iMAX BPM Processes
(Ref 25:BPM-2)

Attribute	Ada tasks	iMAX BPM processes
Scheduling	Advisory priorities fixed at compile-time	User can dynamically vary: priorities, deadlines, time slice length, and number of time slices before rescheduling consideration
Hierarchy	Implied by nesting of declarations. When a task is aborted, any dependent tasks are aborted.	Processes can be organized into trees and process operations can apply to entire trees
Control	Abort another task, raise Failure exception in another task	Start, Stop, Reset, Restart, and Destroy other processes. Use guardian ports to receive and restart processes suspended by some condition.
Communication	A task can wait for multiple entries guarded by conditions. Timeouts can be included in the alternatives waited for.	Processes send or receive messages via explicitly-identified ports. Surrogate operations support prioritized queuing of messages and waiting for the occurrence of one of several different events.
Mutual Exclusion	Not explicitly provided, but can be constructed using communication facilities.	Not explicitly provided, but can be constructed using communication facilities.
Portability	Ada tasks are part of standard Ada.	iMAX processes are not part of standard Ada.

two ways; Ada tasking facilities and the iMAX Basic_Process_Management package (BPM). The facilities provided are not the same but, they may be used together or separately, as the design requires. Table C-I compares the BPM features with Ada tasking. At the time of this writing, the Ada compiler does not support tasking. However, when tasking is supported, design decisions, relating to which management system should be used, may be based on this comparison. Details of the Ada tasking facilities can be found in the Reference Manual for the Ada Programming Language (Ref 20:9-1/9-16). Note that it is advised not to intermix the task control features of Ada and iMAX because there are minor differences in the task information structures used in each system. There are cases where the results of intermixing the operations will cause unpredictable results.

The Basic_Process_Management package (BPM) provides the process management functions required to complement the hardware features of the Intel 432. Essentially, BPM provides the system designer with an interface to processes and their scheduling at a low level without interfering with the system objects used by the hardware (Ref 25:BPM-1). This interface includes manipulation of multiple processes in tree organizations, handling processes unable to execute through special user-defined inter-process communication ports, process operations for creation, control, and

destruction of processes, as well as procedures for setting and examining process attributes and scheduling parameters. The following paragraphs discuss these significant features of the BPM package:

iMAX allows processes to be organized into tree structures which may be controlled or destroyed as a single unit. The organizational placement of a process is determined when the process is created. In simple terms, if the process is created from the "local" memory heap of another process, then it is the child of that process. If the new process is created from the "global" memory heap, then it is not a child of any other process. After creation, BPM control procedures provide for starting, stopping and destroying single processes or entire tree structures. In addition, whenever a process terminates at completion, is stopped or destroyed by action of another process, cannot handle an error condition, or needs service for any other reason, the process is sent to a special communications port defined as the guardian for that process.

The guardian port identified when the process is created. Each dynamic process must have a guardian port. Each guardian port must have an "owner" process which receives the processes sent to the guardian and services each according to its condition. In this way, processes may be destroyed or error conditions may be corrected after the

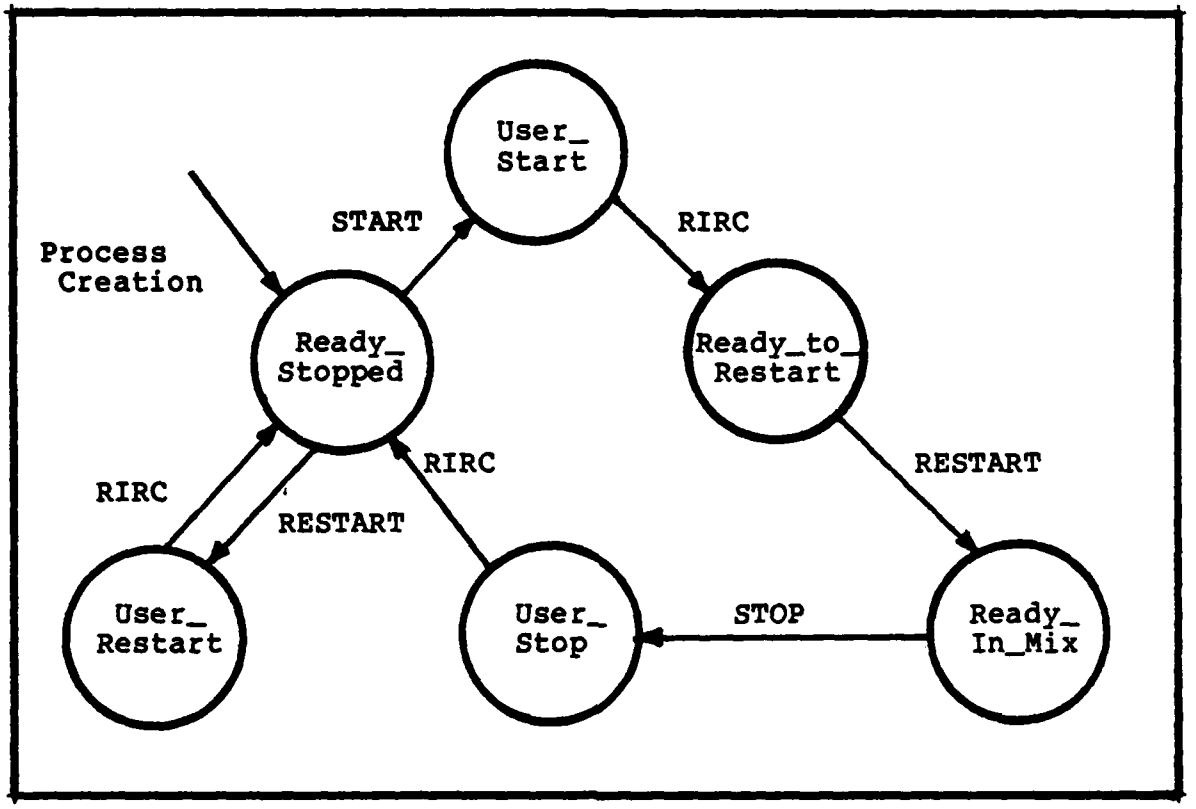


Figure C-2. Dynamic Process State Transitions with BPM (Ref 14:BPM-12)

process is sent to the guardian. It is the user's responsibility to identify the guardian port and the owning process while the system handles moving the process to the guardian port when its condition cannot be satisfied by normal processing.

The condition of a process is identified by its attributes. There are six attributes defined for each process: process name, identification number, trace condition (boolean), guardian port, process globals access segment (a table of process objects), and the process state.

Only the process globals access segment and the process state can be read by the user after the process has been created. The process state is changed as the process is moved through the cycle of scheduling states. Figure C-2 shows the process state transitions for dynamic processes using the BPM package. In that figure, "RIRC" refers to the Read_Info_and_Reset_Condition procedure which is the only way that an existing process may be placed in the READY state (Ref 25:BPM-15).

Storage Management

This section describes the Storage Management features of iMAX. Generally, the concepts of Ada storage management are supported by iMAX (i.e., the access types and allocators used by the Ada new operator) (Ref 25:STO-1). Table C-II summarizes the capabilities of iMAX and the limitations of the current system (Version 2). The major elements of the Storage Management system are Storage Resource Objects (SROs), Lifetime Strategies, Fragmentation and Compaction, and Memory Types. These elements are described briefly in the following paragraphs:

The SROs are the users access points to the memory of the system. The SRO coordinates the use of object table entries and the allocation of physical storage, to provide the user with an, apparent, unbounded claim to memory. When the user requests additional storage, and either virtual address space (an object table entry) or physical space (a

TABLE C-II

iMAX 432 Storage Management Capabilities
(Ref 25:STO-1/STO-2)

iMAX V2 provides a real-memory system with:

1. Dynamic allocation of objects
2. Transparent expansion of object tables and stack or heap storage blocks, as required by user processes
3. Storage reclamation transparent to users
4. A range of lifetimes for created objects

iMAX V2 does not support:

1. Virtual Memory
2. Limits on the amount of memory used by a particular process or collection of dynamically allocated objects

physical memory partition) is not available, then iMAX allocates more resources to the user, transparently. Each SRO, also, has a particular lifetime strategy and a memory type for objects allocated from it.

There are three storage lifetime strategies; stacks, global heaps, and local heaps. The stack lifetime strategy is the most restrictive. Stack objects are deallocated when the context, that created them, is no longer active (i.e., upon returning from that subprogram). Global heap lifetime strategy is the least restrictive. Objects created from the global heap can only be deallocated by the "garbage

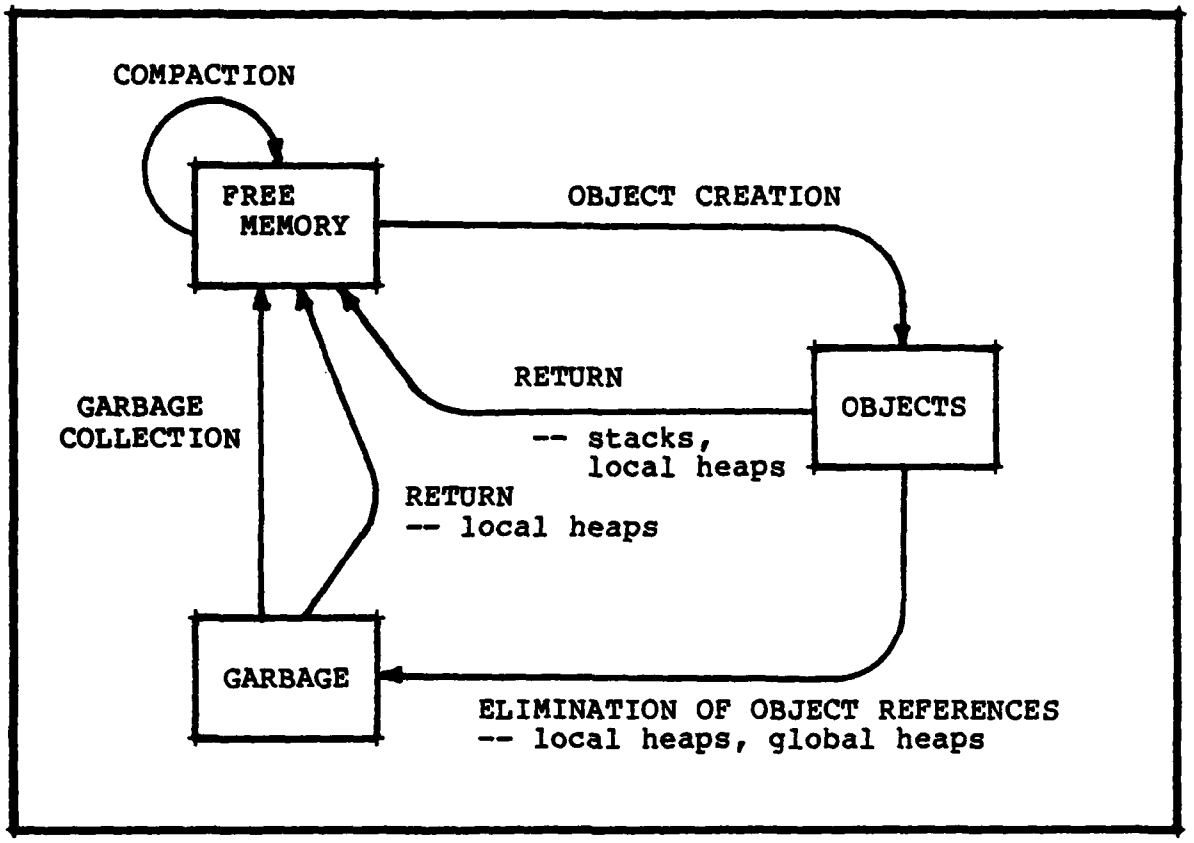


Figure C-3. iMAX Storage Management Transitions (Ref 25:STO-4)

collector" process, which wanders around memory, searching for objects that are no longer in use by any context. This garbage collection process runs concurrently with any other iMAX and user processes in the system. The third lifetime strategy, local heap, is a combination of the other two. If a local heap object becomes unreferenced during its life, it may be reclaimed by the garbage collection process. However, the object is also deallocated on returning from the context that created it. Figure C-3 summarizes the three lifetime strategies by showing the transition paths

of the iMAX storage management system.

The fragmentation of system memory is the division of free physical storage into small portions as a result of allocations and deallocations. iMAX runs a transparent compaction process to reorganize the free memory into larger contiguous segments. It is important to note that the compaction process runs asyncronously and may cause delays accessing objects during its execution. Some parts of iMAX and user programs cannot tolerate this unpredictable delay. Therefore, iMAX divides physical memory into two types; frozen and normal memory.

Segments in normal memory may be relocated by the compaction process. Segments in frozen memory are not relocated by the compaction process. iMAX, Version 2, provides one global heap SRO for frozen memory and one for normal memory. Users may use frozen memory for time-critical processing. However, it must be realized that frequent allocations and deallocations of frozen memory can cause irreparable fragmentation of that part of memory.

Interprocess Communication

This section describes the iMAX facilities for interprocess communication. There are three object types used in interprocess communication; messages, carriers, and ports. The basic operations on these objects are sending a message in a carrier to a port and receiving a message in a carrier from a port. Either of these operations can cause a

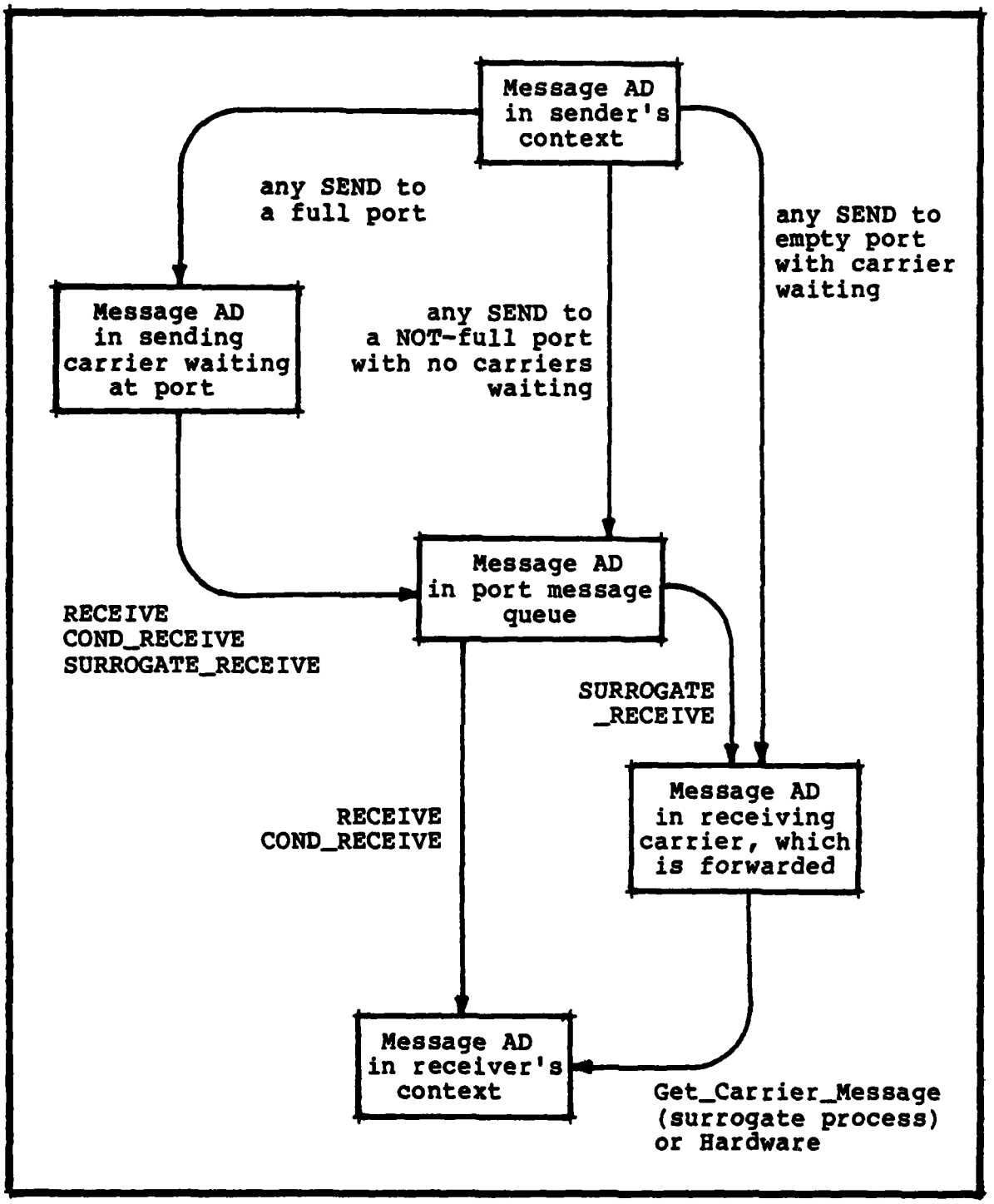


Figure C-4. iMAX Message AD State Transitions
(Ref 25:COM-6)

third operation, the forwarding of the carrier to a second port. These objects and operations are described in the following paragraphs:

Messages are moved by copying access descriptors (ADs) (like pointers). Figure C-4 shows how the reference to a message changes as the AD moves between processes. It is important to note that both the sending and receiving process have access to the message after it has been sent.

In the 432 system, ports are supported by hardware. They consists of two queues; a fixed length message queue and an unlimited carrier queue. The maximum size of the message queue is fixed when the port is created. Messages are placed in the queue according to the port's queuing dicipline (either FIFO or priority). When a message is sent to a port who's message queue is full, the sending process is blocked and the carrier must wait in the carrier queue. When the message queue is empty and the receiving process attempts to get a message from the port, the receiving carrier must wait in the carrier queue. At the time a port is created, the length of the message queue and the queuing dicipline are set, and the queues are initialized as empty.

Carriers move messages to and from the ports. Each process has a process carrier that transports the process around the system ports. Forwarding a process carrier to a dispatching port, allows that process to run. Users can also create surrogate carriers, which are used to move

messages between processes. Surrogate carriers may, also, have a priority which determines where the message is placed in the queue at a port with a priority message queue dicipline (Ref 25:COM-3).

There are three operations which may be used to send messages. Send uses the sending process's carrier to move the message to the receiving port. Surrogate Send uses an specified surrogate carrier. Both of these operations will always result in a message being sent. The third operation, Cond Send, only moves the message if the receiving port is not full. That is, Cond_Send will return a false indication to the sending process if the carrier would have otherwise been blocked at the receiving port. If the receiving port is full, Send and Surrogate_Send will cause the carrier to be placed in the receiving carrier queue and, thus, block further execution of the carrier's process (Ref 25:COM-4).

Similarly, there are three receiving operations. Receive and Surrogate Receive cause their respective process carriers to receive a message from the specified port. If no message is waiting at the port, the carrier is placed in the queue and the process is blocked. The Cond Receive, however, operation returns a false value if there is no message waiting at the port (Ref 25:COM-5).

Input/Output

This section describes the iMAX facilities for data transfer between peripheral devices. There are two general

TABLE C-III

Syncronous I/O Interface Operations and Device Types
(Ref 25:IO-4)

Operations	Device Types		
	Source	Sink	Store
Interface_description	X	X	X
Close	X	X	X
Reset	X	X	X
Transform_interface	X	X	X
Get_asyncronous_ interface	X	X	X
Flush		X	X
Read	X		X
Write		X	X

X = operation is available with device

interfaces provided; a syncronous interface and an asyncronous interface. The syncronous interface facilities can be used to "implement higher-level facilities, such as the Ada TEXTIO package" (Ref 25:IO-1). The asyncronous interface facilities are intended to be used only to implement device drivers or special I/O requirements. The iMAX I/O model supports both new devices and new types of devices. The model does not support the creation or deletion of devices (or device interfaces) during program execution. Also, iMAX does not manage concurrent access to shared devices (e.g., if two processes writing to a shared printer, the printer use needs to be coordinated).

The syncronous interface consists of several packages of software that include a standard procedural interface,

an extension to that interface to support terminal devices, and an instance of the standard interface to support the 432 Debugger system. The standard procedural interface includes definitions of three generic devices in the iMAX I/O model; a source (input-only) device, a sink (output-only) device, and a store device. Table C-III shows the I/O interface operations and the generic devices to which they apply. A detailed description of the operations (and the special packages for terminal and Debugger I/O) can be found in the iMAX Reference Manual (Ref 25:IO-1/IO-15). Note that, for all device interfaces provided by iMAX, Version 2, the Close and Reset operations perform no function (Ref 25:IO-4).

The asyncronous interface facilities define a port-based connection between the device and the processes requesting I/O services from it. A process sends a request message to the port and, later, receives a reply message from the device indicating the success or failure of the operation. iMAX standardizes the interface by defining the format of both connections and the I/O messages. Also, the command codes and reply codes that are used in the messages, are defined by the iMAX facilities (Ref 25:IO-16). Table C-IV shows the relation between the interface commands and the replies they may receive.

The connection is defined by the data in the connection record; an access for the receiving port, a printable name for the connection, an access for the data which describes

TABLE C-IV

Asyncronous Interface Command and Reply Cross-Reference
(Ref 25:IO-20)

COMMAND: / REPLY:	reset	read	write	close	flush	set_device characteristics	get_device characteristics
success	X	X	X	X	X	X	X
end_of_file		X					
not_processed		X					
reset_required		X	X	X	X	X	X
invalid_command		X	X		X	X	X
bad_data_ buffer_size		X	X				
invalid_request	X	X	X	X	X	X	X
hard_IO_error		X	X				
interface_closed	X	X	X	X	X	X	X
reset_returned		X	X	X	X	X	X

X = command can get reply

the device, and an access for the port at which reply messages can be received. The reply port access is also a part of the message format.

An I/O message is an access for a record containing an access to a command record for the message and an access to the reply port, where the message will be returned. The command record contains the command code, the message identifier (allows the reply to be matched with the message sent), the reply code (filled in by the device), and the buffer descriptions (in iMAX, Version 2, there is only one buffer permitted with each I/O message). The command codes and reply codes are explained in Chapter IO of the iMAX Reference Manual (Ref 25:IO), but, generally, their meaning is evident from the name (see Table C-IV).

System Configuration

This section discusses the configuration of the 432 system using the iMAX facilities. The iMAX user can control three aspects of the configuration; the number and identification of the system processors, the static user processes (discussed previously), and the I/O device interfaces present in the system. The configuration of static processes was discussed in the first section of this appendix and the I/O devices were, also, discussed earlier. This section will discuss the configuration of processors in the 432 system.

The amount and type of memory available in the system

Slot No.	System Use	Default
12	GDP or IP	IP
11	GDP or IP	IP
10	GDP or IP	empty
9	GDP or IP	GDP
8	GDP or IP	GDP
7	MEMORY CONTROLLER	MC
6	STORAGE ARRAY	SA
5	STORAGE ARRAY	SA
4	STORAGE ARRAY	SA
3	STORAGE ARRAY	SA
2	STORAGE ARRAY	SA
1	STORAGE ARRAY	SA

Figure C-5. 432/670 System Bus Hardware Configuration

is determined by directives to the linker program (LINK432). The linker is described in the Intel 432 Cross Development System VAX/VMS Host User's Guide (Ref 21:3-1/3-41).

There are three types of procesors in the System 432/670; General Data Processors (GDPs), Interface Processors (IPs), and Attached Processors (APs). The GDPs are the primary processing elements of the system. The APs are the I/O processors and handle the direct interface with the system's peripheral devices. The IPs are the

communications processors that handle information passing between the GDPs and APs. Appendix A explains the system architecture in more detail.

The system configuration information resides in package bodies which the user may alter. The user can modify, recompile and then replace the Ada configuration package body using the 432 Ada Compiler System and LINK432 facilities. The software configuration package contains calls to iMAX procedures which create processor objects. Each functioning processor requires a processor object and iMAX system initialization processes start all processors for which there are processor objects. Since, there is only a small space penalty for configuring more processors than necessary, the system can be configured for the maximum number of processors possible. Then processor boards may be added or removed, as necessary, without any change in the software (Ref 25:CON-1/CON-2).

There are five processor slots in the 432/670 system chassis. These slots can be used for either GDPs or IPs, as needed. However, there must be at least one GDP and one IP present. Thus the maximum number of GDP processors in the AFIT/ENG 432/670 system (with a 12 slot system bus backplane) is four, with one IP. Figure C-5 shows a 12 slot backplane and the slot allocations (Ref 25:CON-2,HDW-1).

Summary

This appendix has outlined the major features of the iMAX 432 Multifunction Applications Executive for the Intel 432/670 computer system. The areas discussed included process management, storage management, interprocess communication, input/output, and configuration of the system in hardware and software. The iMAX Reference Manual (Ref 25) should be used if greater detail is required.

APPENDIX D

Message Format for a Computer Based Message System

Introduction

The purpose of this appendix is to provide a reference document for the specification of the message format for the I/O Interface. The format of the messages used in the interface meet the requirements for message format described in the Proposed Federal Information Processing Standard "Specification for Message Format for Computer Based Message Systems" published by the National Bureau of Standards (NBS), Institute for Computer Sciences and Technology (Ref 38). System users should refer to that document for a complete description of the message NBS standard message format.

As described in Chapter II of the body of this thesis, an I/O Interface message contains six fields; FROM, REPLY-TO, POSTED-DATE, TO, SUBJECT, and TEXT. Each field contains one, or more, data elements, which contain four parts; the identifier, length, qualifier, and data contents. Figure D-1 shows this structure. The purpose of this appendix is to define the information required in each of these components for the I/O Interface messages.

This report presents the structure of the I/O Interface messages in three parts. First, the I/O message is defined by describing the fields contained in the message. Second,

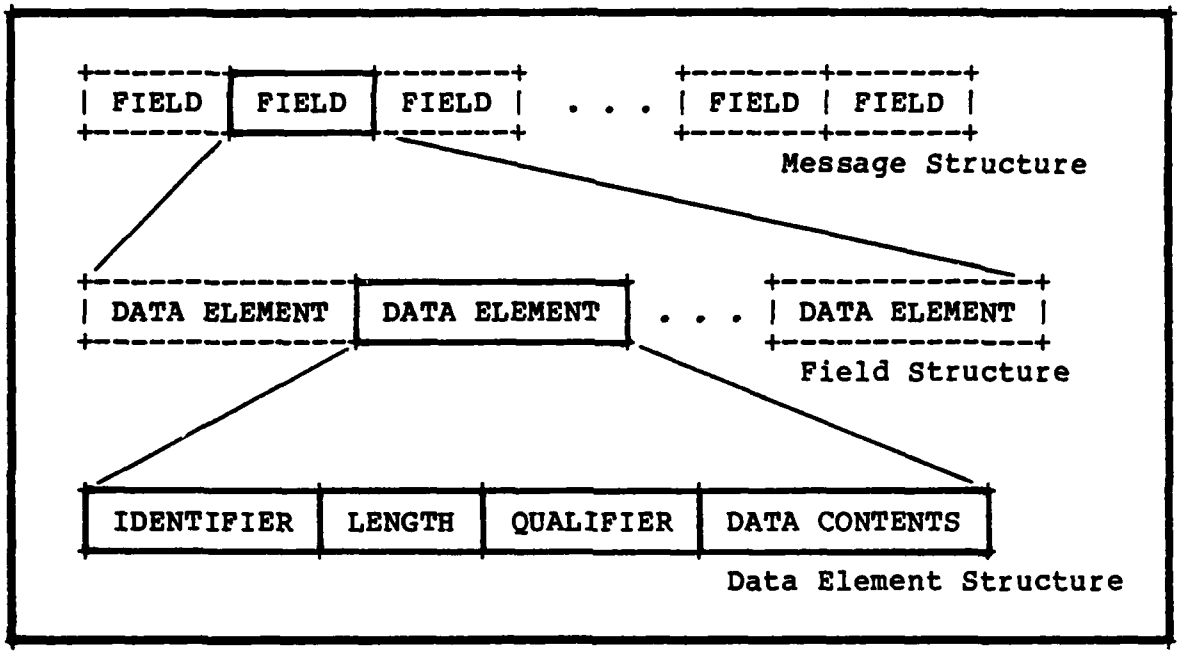


Figure D-1. General Message Structure

the I/O envelope is described, including its relation to the I/O message. Finally, the mapping of I/O device names to CBMS addresses is presented, for reference.

In the following discussions, the words "byte" and "octet" are both used to mean "an 8-bit" value. There is no difference in meaning intended by the use of either word.

I/O Message Description

The I/O message is element of communication among the I/O Interface system User Agents and Device Agents, as described in the body of this report. These messages contain the six fields shown in Table D-I. Messages, and fields, are constructed from data elements. In fact, messages and fields are just special types of data elements.

TABLE D-I

I/O Message Fields

Field	Description
FROM	Identifies the sender
REPLY-TO	Identifies where the reply should be sent
POSTED-DATE	Identifies the date the message was sent
TO	Identifies the recipient
SUBJECT	Identifies the message purpose
TEXT	The content of the message

Each data element consists of four components; an identifier octet, a length code, a qualifier, and the data contents. The identifier octet is a unique 8-bit value that identifies the data element. The length code indicates the number of bytes following it in the data element (i.e., excluding the identifier octet and the length code itself). The qualifier is included to provide information necessary to the interpretation of the data contents. For example, if the data element is a field, the qualifier could contain the code identifying the particular kind of field. Finally, the data contents component of the data element is the actual data represented by the element. The length of the data contents is determined by the difference between the value of the length code and the length of the qualifier component.

The length code and qualifier are, generally, only one

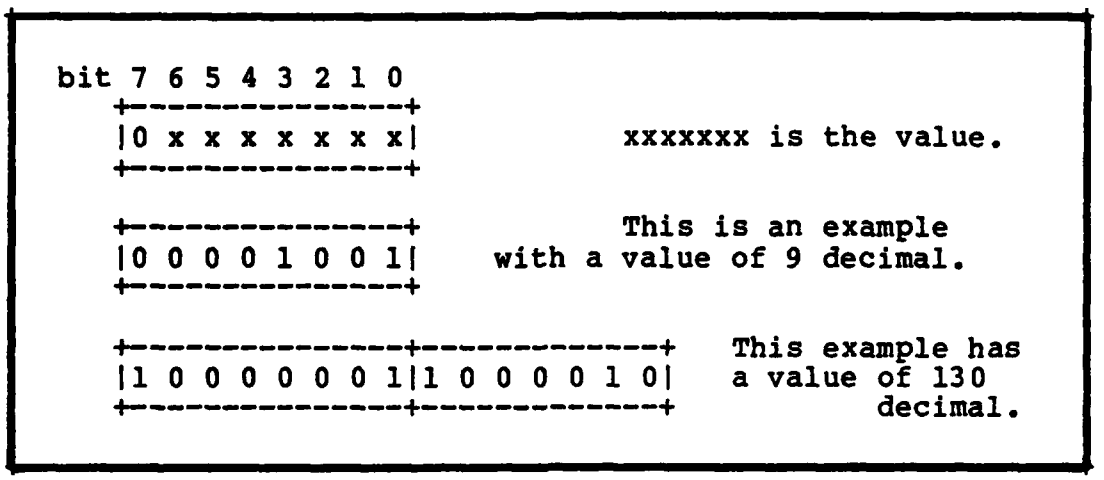
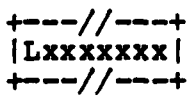


Figure D-2. Encoding Mechanism for Qualifiers and Length Codes (Ref 12:40)

byte in length. However, they are encoded to allow extended lengths to the components. The most significant bit of the components is used to indicate whether the component is one, or several, bytes in length. If the most significant bit is 0, then the remaining 7 bits are the actual value of the length code or qualifier (i.e., in the range 0-127). However, if the most significant bit is a 1, then the remaining 7 bits of the first byte are the number of bytes in the rest of the component. The actual value of the component begins in the next byte and is interpreted as an unsigned integer. Figure D-2 shows examples of this encoding.

For the remainder of this appendix, the following notation will be used to show a length code, where the actual value is unknown:



A similar symbology will be used to denote the other components of the data elements when necessary.

Construction an I/O message begins with the identification of the structure as a message. All I/O messages are identified as NBS standard messages by the following format:

```
identifier length  qualifier  data contents
+--------+---//---+--------+--------//--------+
|01001101|Lxxxxxxx|00000001| message contents |
+--------+---//---+--------+--------//--------+
```

The length code value is 1 more than the actual length of the message contents and the "message contents" consists of the fields listed in Table D-I.

The FROM field in the I/O message contains the I/O Interface device name. The device name is an ASCII string of 17 characters. The NBS standard format specifies the encoding for this field as follows:

```
identifier length qualifier identifier length  data contents
  field      20     FROM       ASCII      17
+--------+--------+--------+--------+--------+------//------+
|01001100|00010100|00000001|00000010|00010001| device name |
+--------+--------+--------+--------+--------+------//------+
```

The REPLY-TO field contains similar data, the device name for the recipient of the reply message. Thus, the field appears as follows:

```
identifier length qualifier identifier length  data contents
  field      20    REPLY-TO   ASCII      17
+--------+--------+--------+--------+--------+------//------+
|01001100|00010100|00000011|00000010|00010001| device name |
+--------+--------+--------+--------+--------+------//------+
```

The POSTED-DATE field contains the date and time of the message being created. This information is formatted in NBS standard date structure (Ref 12:51). This is, simply, the "year, month, day" sequence represented by eight ASCII digits. For example, January 21, 1981, would appear as "19810121". The NBS format of the POSTED-DATE field is as follows:

identifier field	length 13	qualifier POSTED-DATE	identifier date	length 10
01001100	00001101	00000010	00000010	00010001

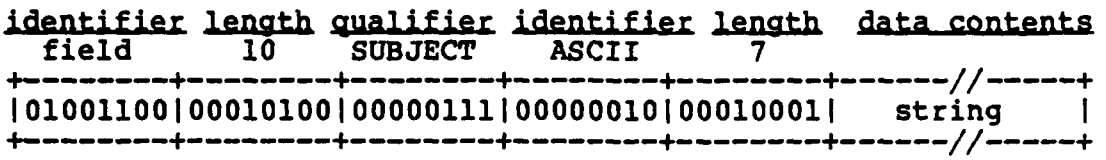
identifier ASCII	length 8	data contents
00000010	00001000	xxxxxxxx

The TO field, like the FROM and REPLY-TO fields, contains an I/O Interface device name. So, the structure is similar to those previous fields:

identifier field	length 20	qualifier TO	identifier ASCII	length 17	data contents
01001100	00010100	00000101	00000010	00010001	device name

In the NBS standard, the SUBJECT field usually contains comment-like data giving some information about the data in the message (Ref 12:62). For the I/O Interface, this field is used to indicate whether the message is an I/O command or a reply from a device. The contents of the field are not used by the I/O Interface system, as it is presently

implemented. However, the CBMS Manager package fills in the field with the proper string (either "COMMAND" or "IOREPLY") and the value may be used, during debugging, to identify reply messages in the system. The field has the following structure:



Finally, the TEXT field contains the data of the I/O message. There are five data elements in this field; the command code, the reply code, a file name, a maximum buffer size, and the data of the message. All five elements must be present in the I/O message, although, their lengths may be zero.

The command code and reply code are integer values, in the range from 0 to 128. Thus, they can be encoded in a single byte.

The file name is an ASCII string of not more than 16 characters. When the file name is not required, the data element's length code is set to 0.

The maximum buffer size is used with the "read" command, to indicate the maximum amount of information that can be returned in the reply message. The buffer size is given as an unsigned 8-bit integer.

The message text is designated as ASCII, although, any 8-bit structured information may appear in the message. The

length of the text is specifically limited. However, the length of the entire message may not exceed 1024 bytes. This limit is implemented in the I/O Interface to simplify the message handling procedures with the current Ada compiler's language restrictions.

These five elements can make the complete message range in size from 105 bytes to the maximum (1024 bytes). The NBS format for the TEXT field is:

To summarize the format of the I/O message, an example is given. The following diagram is a an I/O message from the User Shell process, to the Printer System device, written on August 16, 1983, which is a command to open the printer device for use by the User Shell:

identifier	length	qualifier
message	108	NBS std
01001101	01101100	00000001

identifier	length	qualifier	identifier	length	data contents
field	20	FROM	ASCII	17	
01001100	00010100	00000001	00000010	00010001	RM67/NET0/432/USR

identifier	length	qualifier	identifier	length	data contents
field	20	REPLY-TO	ASCII	17	
01001100	00010100	00000011	00000010	00010001	RM67/NET0/432/USA

identifier	length	qualifier	identifier	length
field	13	POSTED-DATE	date	10
01001100	00001101	00000010	00000010	00010001

identifier	length	data
ASCII	8	
00000010	00001000	19830816

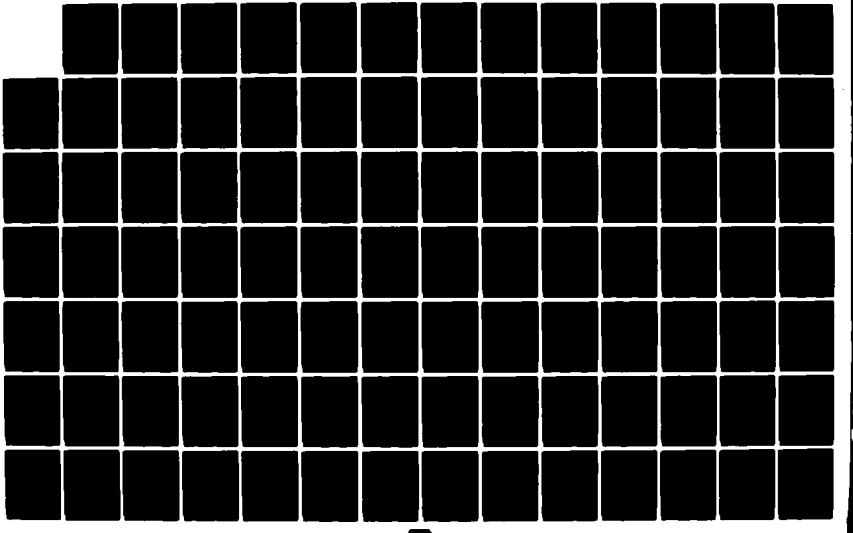
identifier	length	qualifier	identifier	length	data contents
field	20	TO	ASCII	17	
01001100	00010100	00000101	00000010	00010001	RM67/NET0/MDS/PTR

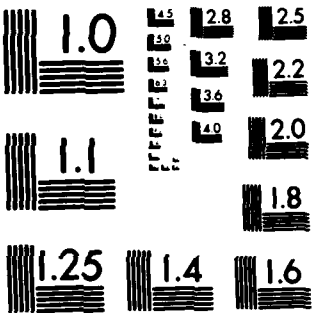
identifier	length	qualifier	identifier	length	data
field	10	SUBJECT	ASCII	7	
01001100	00010100	00000111	00000010	00010001	COMMAND

AD-A138 429 DESIGN AND IMPLEMENTATION OF AN INPUT/OUTPUT INTERFACE PROTOCOL FOR THE I..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. K N COLE

UNCLASSIFIED DEC 83 AFIT/GE/EE/83D-17 F/G 17/2 NL

3/4





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

identifier field	length 12	qualifier TEXT
01001100	00001100	00000100

identifier integer	length 1	data command	identifier integer	length 0
00100000	00000001	00000000	00100000	00000000

identifier ASCII	length 0	identifier integer	length 0	identifier ASCII	length 0
00000010	00000000	00100000	00000000	00000010	00000000

This example shows a message of the minimum size. The message structure is intended to be intrepreted sequentially. There is no predetermined index to the position of a field in the message.

I/O Envelope Description

The envelope of the I/O message is not, actually, an enclosure of the message. Instead, it is the addition of information required by the Message Transfer System, to properly handle the CBMS message. For the I/O Interface system, the additional information is the CBMS address of the destination device. This 32-bit, binary address is included in the I/O message, as an additional data element, of each the device name fields (FROM, TO, and REPLY-TO).

The introduction of the data element requires modification of several of the length entries. The CBMS address element is six bytes long (identifier, length, and

TABLE D-II

I/O Interface Device Names

User Process	Full Device Name
User Shell Receive	RM67/NET0/432/USR
User Shell Reply	RM67/NET0/432/USA
Printer System	RM67/NET0/MDS/PTR
ISIS File System	RM67/NET0/MDS/DSK
Series III Console	RM67/NET0/MDS/CON

four bytes of data) and is included immediately after the device name string of each address field. The length entries for each device name field must be increased by six and the message length, therefore, increases by twelve. The format of the CBMS address data element is:

identifier	length	data contents
bit-string	4	32-bit string
|01000011|	|00000100|	|xxxxxxxxxxxxxxxx|

I/O Interface Device Address Mapping

This section identifies the device names to be used in the I/O Interface system and defines their mapping to CBMS addresses. The device naming structure is presented in Chapter III of the body of the report. The format of the DELNET addresses (used as CBMS addresses) is explained in Appendix E. Users should refer to these texts for more information.

There are five user processes, or devices, in the I/O

TABLE D-III

CBMS Address Field Values

CBMS Address Field	Name	Binary	Decimal
Control	-	0000	0
Country	RM67	0001	1
Network	NET0	0000	0
Host	432	0100 0000	64
	MDS	0100 0001	65
Port	USR	0000 0000 0000	0
	USA	0000 0001 0000	1
	PTR	0000 0000 0000	0
	DSK	0000 0001 0000	1
	CON	0000 0010 0000	2

Interface system. Each device name is associated with a particular user process. The device names and their processes are listed in Table D-II.

There are five fields in the CBMS address, corresponding to the four parts of the device name and a control field (see Appendix E). Table D-III shows the field values for each of the elements of the CBMS address. Note that the User field contains a 4-bit filler on the right. The User Code is contained in the left-most 8 bits of the field. The fields are combined to form a 32-bit value for each device process. For example, the Printer System device (RM67/NET0/MDS/PTR) has the address:

0000 0001 0000 0100 0001 0000 0000 0000

Summary

This appendix has presented the format of the I/O Interfaces messages and the mapping of the device names to CBMS addresses. The message structure used in the I/O Interface system is a direct implementation of the NBS standard format presented in the Specification for Message Format for Computer Based Message Systems (Ref 12). The CBMS addressing structure is compatible with the DELNET address format described in Appendix E.

APPENDIX E

DELNET Network Addressing Scheme

Introduction

The purpose of this appendix is to present the details of the addressing scheme used in the I/O Interface Computer Based Message System for the Intel 432 Micromainframe Computer System. This is an implementation of the Digital Electronics Laboratory Network (DELNET) addressing scheme as discribed in the UNID design documentation (Ref 32). Since it is intended that the Intel 432 Micromainframe computer system become a part of the DELNET system, the use a compatable addressing scheme is required for network communications (Ref 9:21).

The DELNET addressing structure is primarily an implementation of CCITT X.121 standard and the Internet Header Format used by DARPA. The address format is 32 bits long and contains five fields; Control, Country Code, Network Code, Host Code and User Code.

Note that the DELNET specification refers to the User-id field as a "Port Code". Either terminology can be used, the structure of the field is not different, but, the idea of a "user" is more applicable to the I/O Interface system use.

Figure E-1 shows the address structure and the following sections will present the domain for each field

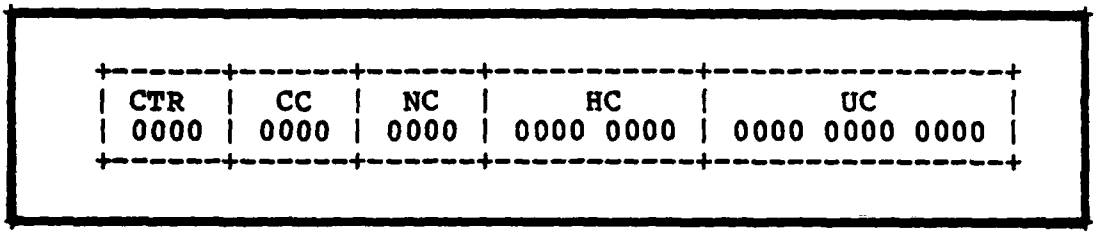


Figure E-1. I/O Interface Message Address Format

and discuss their use in the I/O Interface.

Control (CTR)

This is a 4 bit code which will establish the addressing format to be used. The I/O Interface will only use one addressing format and, therefore, this field may be considered constant. The value [0000] in the Control field indicates that the Country Code field is 4 bits, the Network Code field is 4 bits, the Host code field is 8 bits and the User Code field is 12 bits. This format is shown in figure E-1.

Country Code (CC)

The Country Code is a 4 bit field which gives the first level of addressing heirarchy. The I/O Interface system is located within the same reference area and, again, will only require a constant value for this field. A Country Code of [0001] indicates the system resides in the basement of building 640. If the system is moved or the network is expanded to include a larger geographic area, the specification this field may need to be reviewed.

Network Code (NC)

Within each Country Code area, there may be several networks. The Network Code specifies the next level in the addressing heirarchy. The DELNET specification uses the Network Code to identify a particular UNID in the system, as if a UNID would be used to attach each network to the system. It may happen that several local area networks are attached to a single UNID, due to the limited number of UNIDs available. In this case, modification of the addressing scheme will be necessary. However, like the Country Code, it will still remain constant within the Intel systems implementation of the I/O Interface.

Host Code (HC)

The Host Code specifies a particular computer system within a network. The I/O Interface system is, in fact, a local area network from the viewpoint of the DELNET. The 8 bit code for the Intel systems can range from 0 to 63, [0000 0000] to [0011 1111]. In particular, the 432 processor system is designated Host 0, the Debugger system is Host 1, and the second AP system is Host 2 for the current organization of the Intel systems.

According to the DELNET specification, the Host Code values are dependent upon the hardware port that is used for interconnection with the UNID. When the UNID becomes operational and the systems are interconnected, it may be necessary to modify these values.

User Code (UC)

The User Code is the identification of the particular device or user process which is to be addressed. It is the lowest level in the addressing heirarchy. These 12 bit values may be assigned as fixed values for particular system devices or processes, or they may be allocated during execution and deallocated when not required by the using process or device. Within the I/O Interface system, fixed values will be assigned to each User Agent process. Future implementations of the CBMS may provide other methods.

Summary

This appendix has presented the details of the I/O Interface message addressing scheme. The organization provides for an address space which should be adequate well into the future. The important fields for the I/O Interface will be only the Host Code and the User Code. Constants should be supplied for the upper field values which may be modified at the time the system in connected to the DELNET devices. Further information about DELNET organization and operation can be found in "Protocol Standards and Implementation within the DELNET using the Universal Network Interface Device (UNID)" (Ref 32).

APPENDIX F

Intel 432 Cross Development System Hardware and Software Compatibility Guide

This appendix contains a table of the software and hardware which are compatable for use in the Intel 432 Cross Development System. The configuration presented is for the Release 2 hardware and software which was used during the I/O Interface system development. This is not the latest release of 432 hardware and software from Intel Corporation. Use of the I/O Interface software with a later version of the 432 development environment may require modifications to the interface code modules. Refer to the appropriate Intel documentation for each development system component or utility regarding system upgrades.

All of the information presented here comes directly from Intel publication number 172547-003 "Intel 432 Cross Development System Hardware and Software Compatibility Guide." For more information the reader should refer to that document and the "Intel 432 Cross Development System VAX Host User's Guide" (Ref 21).

TABLE F-I

Hardware and Software Compatibility Guide
for Release 2 Components

Product	Product Code	Version	Replaces
ACS 432 (VAX/VMS)	CDS432-11 R	V1.01*	V1.00
(VAX/Unix)	CDS432-21 R	V1.01*	-----
LINK-432 (VAX/VMS)	CDS432-12 R	V1.01*	V1.00
(VAX/Unix)	CDS432-22 R	V1.01*	-----
DEBUG-432	CDS432-30 T/U	V1.01	V1.00
UPDATE-432	CDS432-30 T/U	V1.00	-----
iMAX 432 (VAX/VMS)	MAX432 T/U		
-432 Code		V2.01*	V1.00
-AP Code		V1.00**	-----
(VAX/Unix)	MAX432 20 T/U		
-432 Code		V2.01*	-----
-AP Code		V1.00**	-----
DSP 432	DSP432-6 A/B	V2.00	V1.XX
iAPX 432 GDP		Release 2.1	Release 1.X Release 2.0
iAPX 432 IP		Release 2.1	Release 1.X Release 2.0

* Compatable VAX Host Operating Systems:

VAX/VMS : Version 2.4
VAX/Unix : Fourth Berkeley distribution of Unix/32V

** Compatible with iRMX 88 V2.0

APPENDIX G

432 Ada Compiler System Version 1.01 Unimplemented Facilities

Introduction

The purpose of this appendix is to present a concise listing of the facilities which are not supported by the current cross compiler system, version 1.01. The information presented here comes directly from Intel Publication Number 172250-005, "432 Ada Compiler System Version 1.01, VAX/VMS Release Unimplemented Facilities," dated 6 August 1982. Additional information on these restrictions may be found in the first appendix to the Intel publication "432 Cross Development System VAX Host User's Guide" (Ref 21:A-1/A-4).

Unimplemented Facilities

The following facilities are not supported in the version 1.01 release of the Ada language cross compiler:

1. Approximate numbers (fixed and floating).
2. Tasking.
3. I/O facilities except the ones specified in the 432 TEXT_IO package.
4. Array operations, notably concatenation and boolean operations.
5. Arrays (including strings) whose bounds are not static.
6. Packed arrays of non-byte-length elements.

7. Arrays of mixed records (records that include access and data values).

8. Arrays longer than 64K bytes.

9. The following pragmas
(Ada language compiler controls):

controlled,
extension,
include,
interface,
list,
memory_size,
optimize,
priority,
rights (for instruction segments),
storage_unit,
suppress,
system.

The "pack" pragma is partially implemented. "Pack" has effect only on new scalar types. For "type s is range 1..10" and "pragma pack (s)" instances of s will be represented by four bits. Without the pragma instances will be represented by 32 bits, the default for universal integers.

10. The following attributes:

address,
constrained,
first_bit,
image,
last_bit,
position,
range,
size,
storage_size,
value,
all fixed and floating attributes,
all tasking attributes.

11. Type conversion involving structural representation changes.

12. Packages with no package body declarations.

13. Run-time checks.

14. The constraint_errors Access_check and Length_check.

15. The following pre-defined exceptions:

    Constraint_error,
    Numeric_error,
    Select_error,
    Storage_error,
    Tasking_error.

16. User-defined exceptions.

17. Dynamic typing as described in Appendix F of the Intel Edition Ada Language Reference Manual (Ref 20).

18. Representations over access types.

19. Record representation "at mod".

20. Address specification.

21. Interrupts.

22. Static packages (i.e., packages not local to a subprogram or block) requiring any executable code for their initialization. Thus, package initialization, initialization of package variables, constants, and separate packages are not supported by the first release of the 432 CDS.

23. Generic Formal types that have discriminates are not checked at instantiation.

24. Access before elaboration checks are not implemented. Recent language discussions suggest that these checks may not be required in the future.

Summary

The facilites which are not implemented or are restricted in use are summarized in Table G-1. While these limitations on the use of the Ada language impose a large burden on the software designer, this was the only compiler available for the 432 system. The latest release of the comipiler, Version 2.0, has many of these problems corrected including arrays and limited tasking facilities. Intel is

TABLE G-I

Ada Compiler System
Implementation Restrictions

(Ref 21:A-1)

Facility	Status
Approximate Numbers	Not Implemented
Tasking	Not Implemented
Packages without bodies	Restricted
Address specifications	Not Implemented
Interrupts	Not Implemented
Change of representation	Restricted
Arrays	Restricted
Run-time checks	Not Implemented
Exceptions	Not Implemented
Representations	Restricted
I/O facilities	Restricted

currently working to complete Version 3.0 which will implement the complete Ada language.

APPENDIX H

I/O Interface User's Manual

Table of Contents

	Page
Table of Contents	H-1
Introduction	H-2
AFIT/ENG 432/670 Development System	H-3
Hardware Systems	H-4
Hardware Interfaces	H-7
Software Utilities	H-9
I/O Interface Configuration	H-11
Hardware	H-12
Software	H-15
Operation of the I/O Interface	H-20
I/O Interface Devices	H-25
Using the I/O Interface Devices	H-27
Printer System Device	H-27
ISIS File System Device	H-29
Series III Console Device	H-31
Adding New Devices	H-32
Operation of the User Shell	H-35
Configuration	H-35
Shell Commands	H-38
Help Command	H-40
Set Command	H-43
Copy Command	H-45
Summary	H-46
Bibliography	H-47

Introduction

The purpose of this thesis (Ref 2) appendix is to provide a structured guide to the use of the I/O Interface for the AFIT/ENG 432/670 Micromainframe Computer System. The following sections describe the operation and use of the I/O Interface, itself, and then the operation of the User Shell software which is an example of an application of the interface.

This document is intended to be used as a user's guide for the I/O Interface. Therefore, there is some duplication of information with other sections of the thesis Design and Implementation of an Input/Output Interface Protocol for the Intel 432/670 Computer System (Ref 2), of which this manual is an appendix. The reader should refer to that document for further information.

AFIT/ENG 432/670 Development System

This section describes the hardware and software environments of the Intel 432 Cross Development System (CDS). The CDS is an interconnection of three computer systems; a VAX 11/780 CDS Host System, an Intel Series III Microcomputer Development System (MDS), and the Intel 432/670 Micromainframe Computer System. Figure H-I shows this organization. The following paragraphs will describe the system hardware, the interconnections, and the software utilities used in each part of the CDS. Users of CDS should

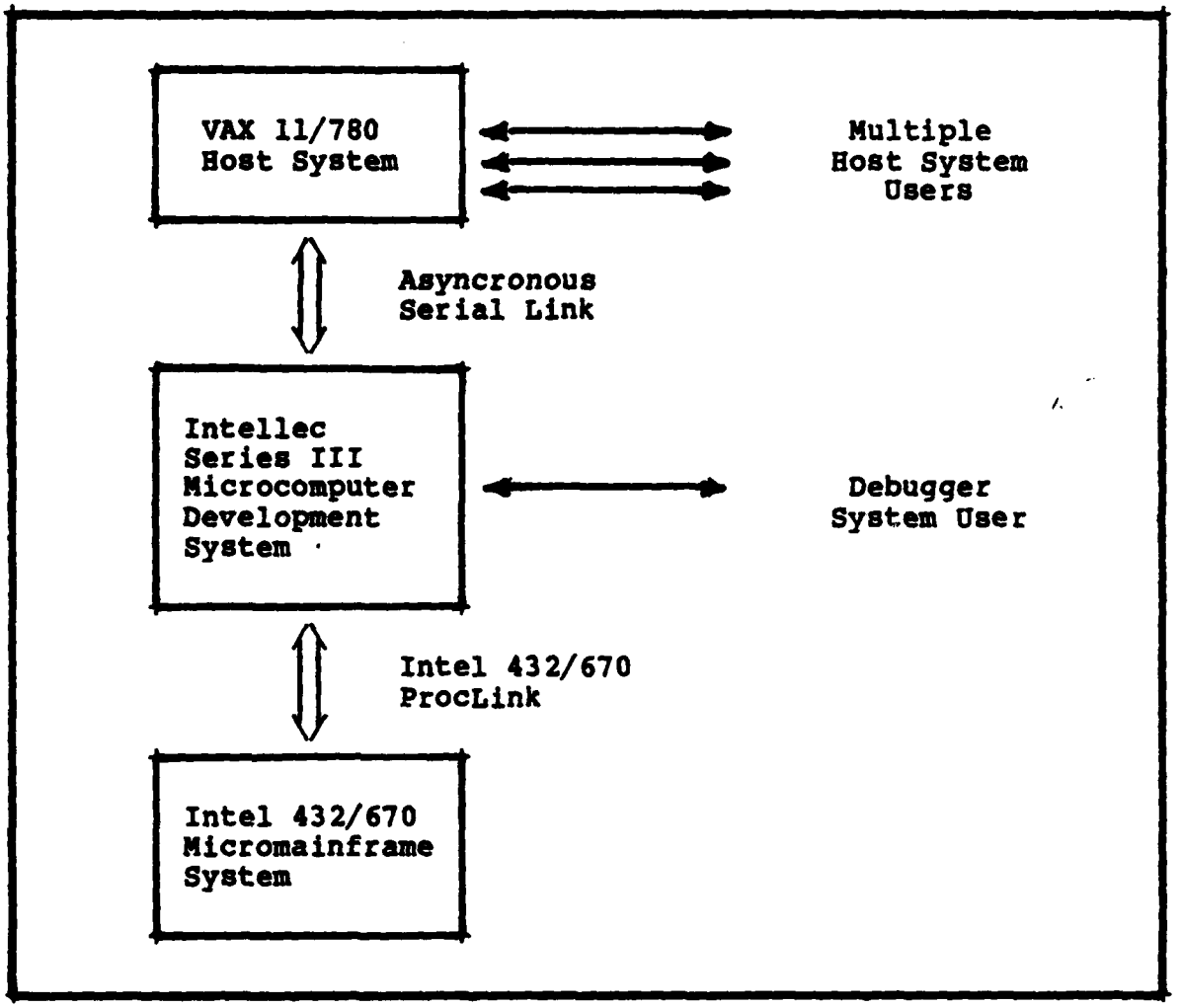


Figure H-1. Intel 432 Cross Development System Hardware Environment

refer to the Introduction to the Intel 432 Cross Development System for a more detailed description of the system (Ref 13).

Hardware Systems. The major components of the CDS are the VAX-11/780 Host system, a Series III MDS Debug Workstation, and the 432/670 System (see Figure H-1). The other necessary hardware includes the communications link

for downloading programs from the host to the Debug workstation, and a 432 Interconnect Kit to connect the Series III MDS to the 432/670 System. The communications link and the Interconnect Kit will be discussed in the next section.

The VAX Host system provides an environment for the software development tools of the Ada Compiler System (ACS) and the 432 linker program. The system also provides access to standard text editors and large data storage systems which can be accessed by multiple users to speed the software development tasks.

The Debug Workstation consists of an Intellec Series III Microcomputer Development System (MDS) connected to both the VAX Host system and the 432/670 computer. The required configuration for the Series III MDS consists of the following hardware:

-- 192K bytes of RAM (minimum)
-- single- or double-density diskette drives
-- one hard disk drive (minimum)
-- an interface to the System 432/670.

The System 432/670 is a flexible computer system that can be configured to meet the processing needs of the user. The maximum configuration includes up to four General Data Processor (GDPs) boards, at least one Interface Processor (IP) system, a Memory Controller (MC) board, and up to six Storage Array (SA) boards providing 1.5 megabytes of memory.

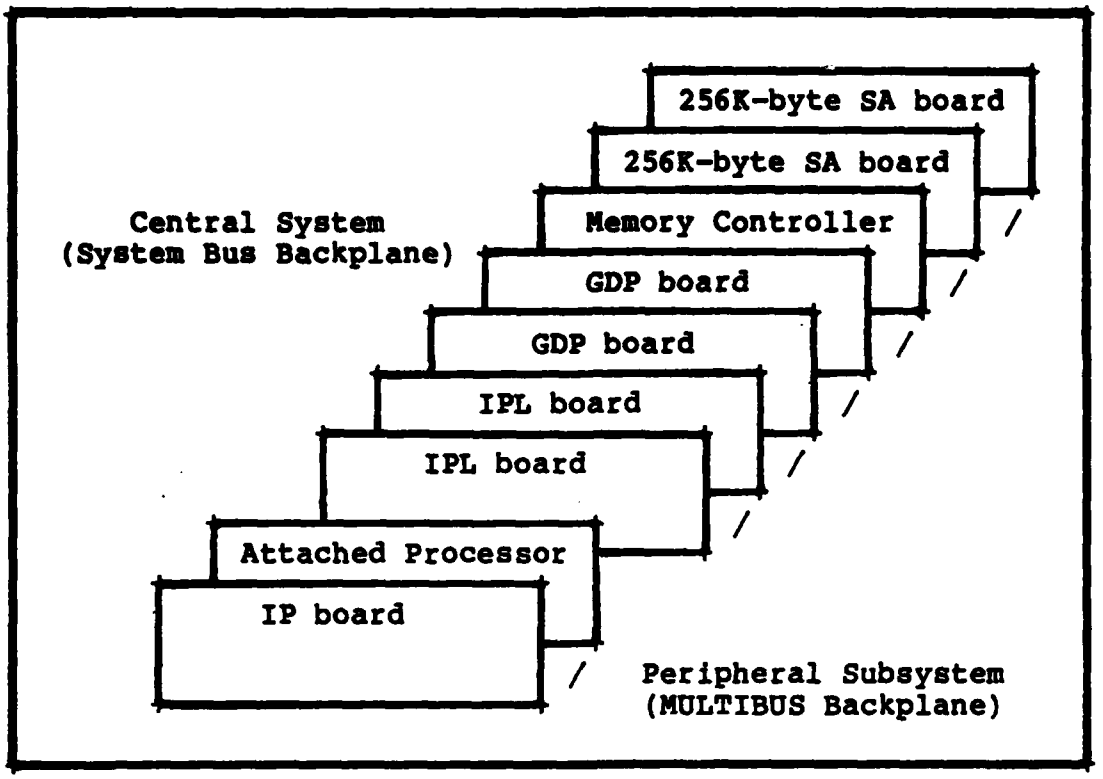


Figure H-2. System 432/670 Standard Configuration (Ref 13:1-6)

The original (factory shipped) configuration of the system includes (Ref 13:1-5):

1. One iSBC 432/630 Enclosed Chassis with a power supply and card cage

2. One iSBC 432/611 12-slot System bus backplane and one iSBC 432/615 6-slot MULTIBUS backplane

3. Two iSBC 432/601 General Data Processor boards

4. One iSBC 432/603 Interface Processor Link (IPL) board

5. One iSBC 432/604 Memory Controller (MC) board and two iSBC 432/607 256K-byte Storage Array (SA) boards

6. One iSBC 432/602 Interface Processor (IP) board

7. One Attached Processor (AP) board (properly configured iSBC 86/12A board) with 32K-bytes of EPROM and 64K-bytes of RAM.

The arrangement of these boards in the 432 system chassis is shown in Figure H-2 (Ref 13:1-6).

Hardware Interconnections. As shown in Figure H-1, there are two major system interconnection in the Cross Development System hardware. The first is the VAX-Series III serial link. The second is the Series III/432 Interconnection, also called the Intel 432 ProcLink.

The VAX-11/780 system is connected to the Intel Series III MDS by an asyncronous serial link. The operation of the serial link is described, in detail, in the Intel manual Asyncronous Communication Link Users Guide (Ref 17). The existing 5-wire unshielded connection runs from the Intel systems, in the basement of building 640, room 67, up to the second floor, room 245, where it is connected to the VAX system. While this connection far exceeds the 50-foot maximum for EIA Standard RS-232-C Type E interfaces (Ref 21:56), the link is sufficient for data transfers at 9600 baud.

The Series III MDS is connected to the 432/670 Micromainframe by the Intel ProcLink cable which provides the data path for the 432 Interface Processor (IP) and the Interface Processor Link (IPL) communication system. This hardware is called the "Intel Series III/432 Interconnect Kit" and is described in the manual Introduction to the

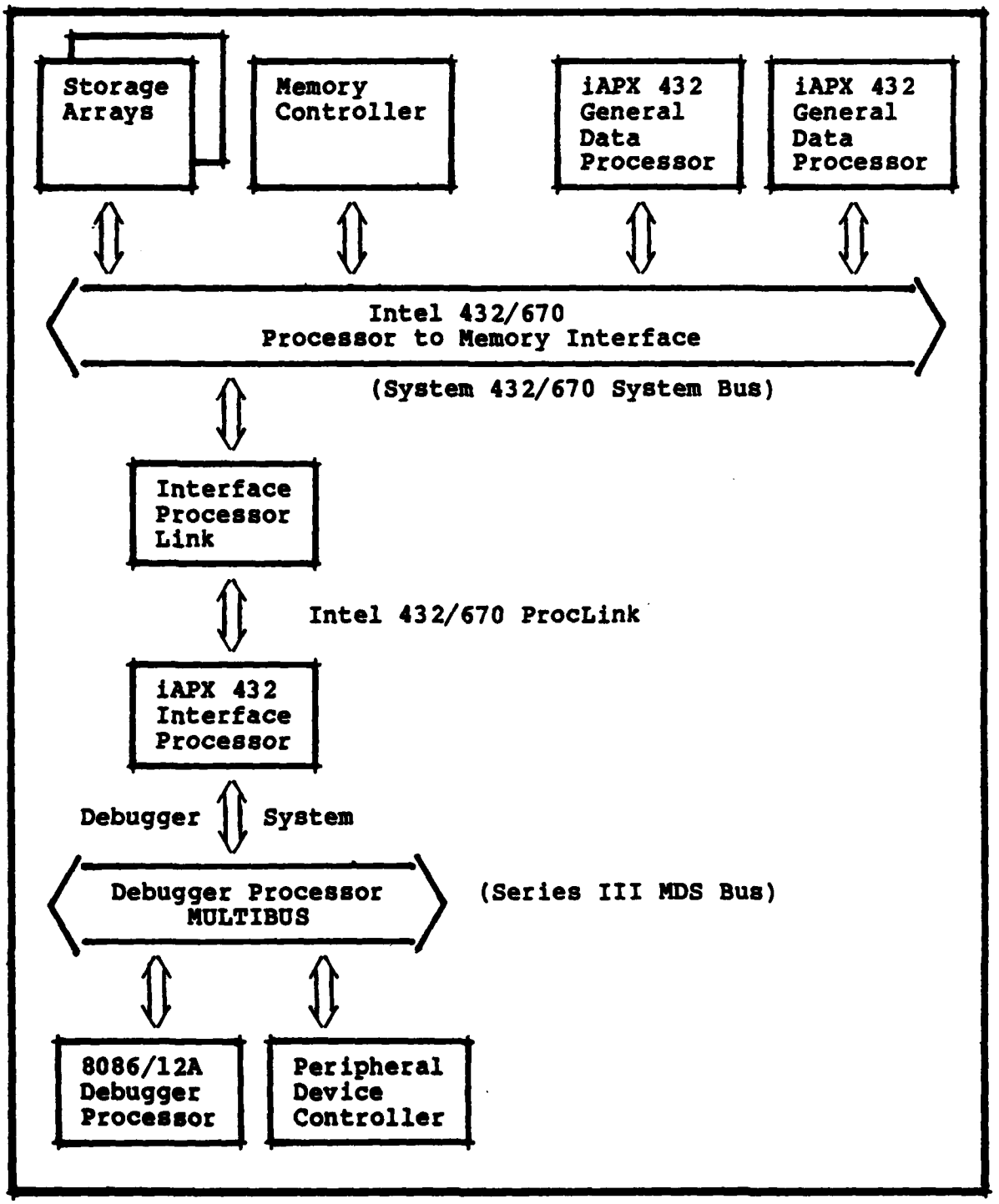


Figure H-3. 432/670 Cross Development System Hardware Configuration

TABLE H-I

432 Cross Development System
VAX/VMS Directories

Contents	VAX/VMS Directory Name
Ada Compiler System (Version 1.01)	[INTEL2.ACS]
iMAX 432 Operating System (Version 2.01)	[INTEL2.IMAX]
432 Link System (Version 1.01)	[INTEL2.LINK432]
Asyncronous Link Software (Version 1.02)	[INTEL2]

Intel 432 Cross Development System (Ref 13:1-7). Figure H-3 shows the ProcLink connection between the IP board on the Attached Processor system bus and the IPL board on the 432/670 system bus.

Software Utilities. The Ada language programming is performed on the VAX host system under the VMS operating system. Software development includes: compiling and revising Ada source programs, compiling programs, creating combined specification files, generating program listings, and linking compiled programs. Standard VAX text editor systems are used to create source files. The CDS host resident software includes the Ada Compiler System (ACS) and the 432 program linker (LINK432) (Ref 13:1-7).

Table H-I shows the directories which contain the CDS software on the VAX/VMS Host system. The files in these directories were supplied by Intel Corporation on 9-track tapes and copied to the VAX/VMS system disk storage according to the instructions in the 432 CDS VAX Host User's Guide (Ref 12:G2-G3). This software is on the "INTEL" labeled disk for the VAX/VMS system in room 245 of building 640.

The Debugger Workstation software tools permit users to handle executable files for the 432/670, load and debug programs on the 432/670 system, and develop device drivers for the 432 Attached Processor. The files necessary for the Debugger Workstation are listed in Table H-II. These program files were supplied by Intel on single-density 8-inch floppy disks in ISIS readable format. For detailed instructions on the operation of the Series III utilities, refer to the Series III MDS Console Operating Instructions (Ref 4) and the Series III MDS Programmer's Reference Manual (Ref 3). Operating instructions for the Debugger, Update and Diagnostic programs can be found in the 432 CDS Workstation User's Guide (Ref 14). These manuals and disk files are maintained, with the Intel systems, in room 67, building 640.

It must be noted that the proper hardware configuration for the Series III Workstation requires a hard disk storage device (Ref 13:1-6). Lack of the larger mass storage device

TABLE H-II

432 Cross Development System
Series III Workstation Software

Contents	File Name
Series III Operating System	(Version 4.3)
Executable Debugger Program	DEB432.86
Intel-supplied Templates	DEB432.TEM
Executable Updater Program	UPDATE.86
Asyncronous Link Software	
Configuration Program	CONFIG
Terminal Emulation Program	ONLINE
One-Way Communication Program	SEND
VAX-to-Series III File Transfer Program	DNLOAD
Series III-to-VAX File Transfer Program	UPLOAD

limits the size of the executable file which may stored on a double-density floppy diskette, 3895 blocks (498,560 bytes). The operation of the system utilities is not affected by this limitation.

I/O Interface Configuration

This section describes the minimum configuration of hardware and software necessary for operation of the I/O

Interface on the AFIT/ENG 432/670 computer system. These configurations do not include the requirements of any applications, which would use the interface, or the Debugger system which may be required for loading and executing the software on the Intel 432/670.

Hardware. The hardware configuration for the I/O Interface operation is an interconnected system of, at least, four distinct microprocessors; the iAPX 432 General Data Processor (GDP), the iAPX 432 Interface Processor (IP), the Intel 8085-based Integrated Processor Card (IPC), and the Intel 8086-based Resident Processor Board (RPB). These processors, and their associated system hardware, are placed on two system bus structures; the Processor and Memory Interface bus of the Intel 432/670 Micromainframe Computer System and the MULTIBUS structure of the Series III Microcomputer Development System (MDS). These two main chassis are interconnected by the Intel ProcLink cable as shown in Figure H-4.

The Processor to Memory Interface bus is the main system bus of the Intel 432/670 Micromainframe Computer System. The 432/670 chassis contains a 12 slot system bus. There are five slots available for processor boards (Ref 13:3-1/3-16). Those slots may contain any combination of GDP and Interface Processor Link (IPL) boards. However, at least one GDP board and one IP Link must be included in the system configuration. The execution environment is defined

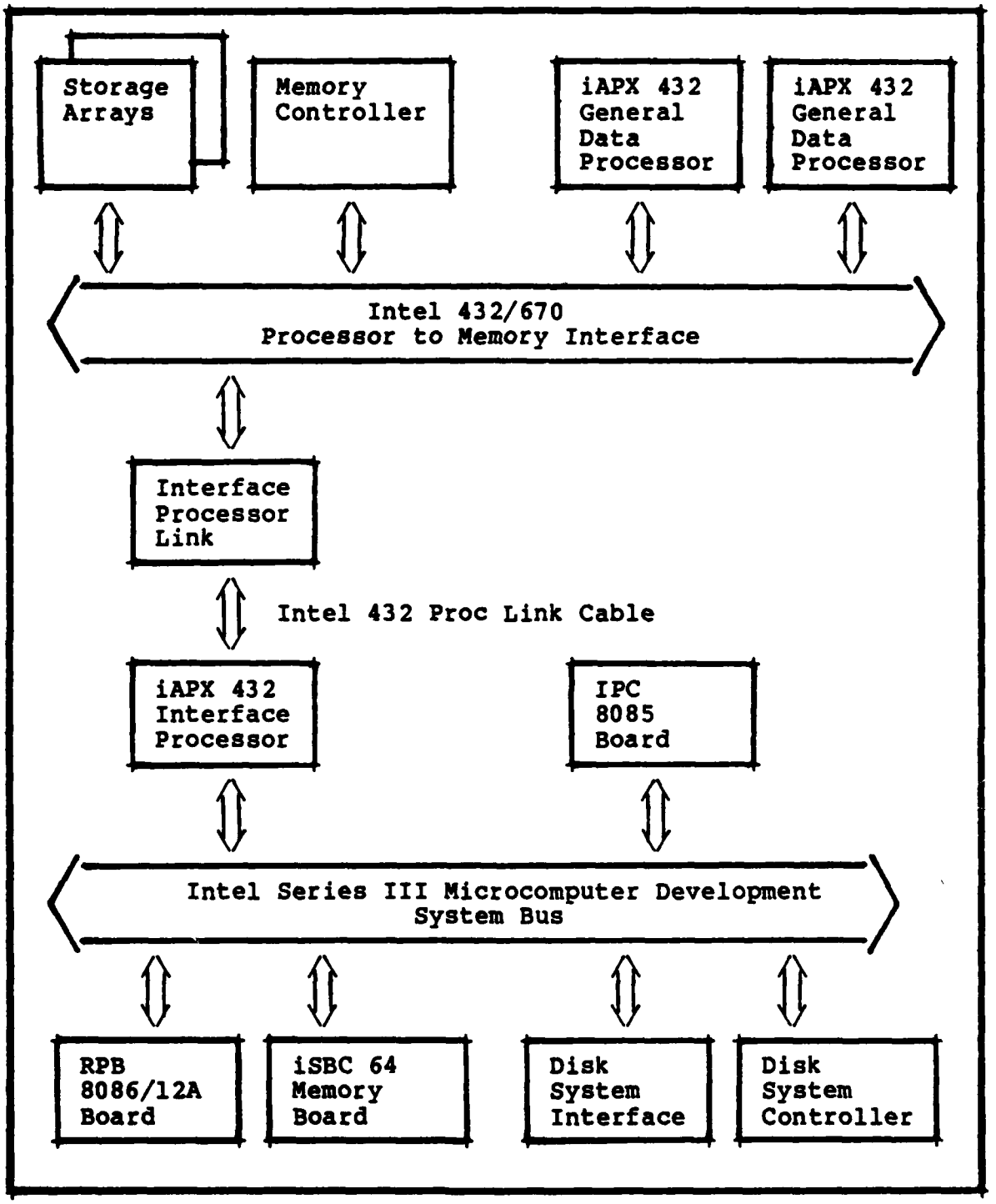


Figure H-4. AFIT/ENG 432/670 Computer System Hardware Configuration

in the Processors package of the iMAX operating system, which must be modified to reflect the current system configuration (Ref 16:CON-2). The minimum configuration of the Processor to Memory bus for operation of the I/O Interface is exactly the configuration of the System 432/670 described with the Cross Development System (see Figure H-2). Additional memory or processors may be required by the application program.

The minimum configuration is also shown in Figure H-4, where the Intel ProcLink cable is shown between the Interface Processor Link board, on the 432/670 system bus, and the Interface Processor board, on the Series III MDS system bus. Figure H-5 also shows the minimum configuration of the Series III MDS. Note that this need not be the same Series III MDS as was used for the Debugger in Figure H-3. In fact, if the Debugger system is required, the I/O Interface AP must be a second Series III system.

The configuration of the MULTIBUS boards in the Series III system depends upon the particular chassis and the optional boards that are included. In general, the IPC-85 processor board is in the top-most slot of the main system chassis and the disk controller/interface boards are placed in the lowest priority slots of the system. For the exact configuration, users should refer to the Intellec Series III Microcomputer Development System Hardware Reference Manual and the System 432/670 System Reference Manual (Ref 15).

Software. The software of the I/O Interface implementation for the AFIT/ENG 432/670 Computer system is organized into packages, or modules, that provide functional implementation of each mechanism of the interface protocol. The following paragraphs describe the software for each system of the 432/670 I/O Interface implementation:

The 432 software packages are written in the Ada language and must be compiled and linked, with the operating system, using the facilities of the 432 Cross Development System. The VAX Host Users Guide provides operating instructions for the Ada Compiler System (ACS) and the 432 system linker (LINK432) (Ref 12:3-1/3-40).

When compiling the source modules, it is important to note the interaction among the packages. Each package must begin with an Environment Pragma which lists the source files which have specifications of modules that are called by the procedures of the package. The source files must be compiled in a sequence that ensures all referenced specifications have already been compiled. Table H-III lists the I/O Interface Ada language files in the order in which they may be compiled. However, several modifications must be made to these files and changes in this sequence may be required.

The processor initialization procedures file (Pserp.mbs) must be modified to properly start the program written to use the I/O Interface. The iMAX 432 Reference

TABLE H-III

I/O Interface 432 Software Packages

File Name	Description
Am.mss	Address Manager Specification
Am.mbs	Address Manager Body
Rm.mss	Route Manager Specification
Rm.mbs	Route Manager Body
Cbms.mss	CBMS Message Manager Spec.
Cbms.mbs	CBMS Message Manager Body
Mts.mss	Message Transfer System Spec.
Mts.mbs	Message Transfer System Body
Usa.mss	User Agent Specification
Usa.mbs	User Agent Body
Pserp.mbs	Processor Initialization Body
Ioface.lkd	Linker Command File

Manual explains the system initialization procedures in detail (Ref 16:INI-1). The tasks of the I/O Interface are "static" processes which started at the time of system initialization (Ref 16:CON-3).

The User Agent package was written for use with a particular user/device package; the User Shell process, which is explained later. To use the I/O Interface with another process, the receive procedure (Usa_receive) of the User Agent package must be modified to work with the functions of that process. In addition, similar User Agent packages must be written for each user/device process which is added to the 432 processor program and requires the services of the I/O Interface. The function of the User Agent is explained in the following section.

When all Ada language modules have been successfully compiled, the resulting 432 object code files must be linked with the operating system module (Imax.eod) using LINK432. The iMAX 432 Reference Manual (Ref 16) contains a complete description of the facilities of the operating system.

The commands for the linker program have been provided in a file (Ioface.lkd). This file must be modified to include the object files containing the program which will use the interface. The linker commands are explained in the Intel 432 CDS VAX Host Users Guide (Ref 12:3-1/3-40).

The 8086 software for the Series III MDS contains a similar set of files. These are listed in Table H-IV.

The Series III MDS software is written in PL/M-86 and utilizes the facilities iMAX provides for process control and memory management. The order in which files are compiled is not important in this environment. Also, the specifications for each package are simply text files which are "included" in the PL/M-86 source files that need to reference procedures in the package. The specification files are designated by the ".inc" extension. The source files (package bodies) have the ".plm" extension. The list includes "System.lib" which is the Series III/ISIS-II operating system library of routines.

These files are compiled using the PLM86 compiler program. Then the iRMX-88 Interactive Configuration Utility (ICU88) is used to identify the operating system modules

TABLE H-IV

I/O Interface 8086 Software Packages

File Name	Description
Am.inc	Address Manager Specification
Am.plm	Address Manager Body
Rm.inc	Route Manager Specification
Rm.plm	Route Manager Body
Cbms.inc	CBMS Message Manager Spec.
Cbms.plm	CBMS Message Manager Body
Mts.inc	Message Transfer System Spec.
Mts.plm	Message Transfer System Body
Psa.inc	Printer System Specification
Psa.plm	Printer System Body
Ifsa.inc	ISIS File System Agent Spec.
Ifsa.plm	ISIS File System Agent Body
S3ca.inc	Series III Console Agent Spec.
S3ca.plm	Series III Console Agent Body
Ps.inc	Printer System Specification
Ps.plm	Printer System Body
Ifs.inc	ISIS File System Specification
Ifs.plm	ISIS File System Body
S3c.inc	Series III Console Specification
S3c.plm	Series III Console Body
Init.plm	Processor Initialization Body
Ioface.lkd	Linker Command File

that need to be included. Finally, all the program and operating system modules are combined by the system linker (LINK86) and an executable file is created. The operating instructions for each of these utilities are provided in the appropriate Intel manuals; the PL/M-86 Compiler Operating Instructions (Ref 20), the iRMX 80/88 Interactive Configuration Utility User's Guide (Ref 5), and the iAPX 86,88 Family Utilities User's Guide (Ref 6).

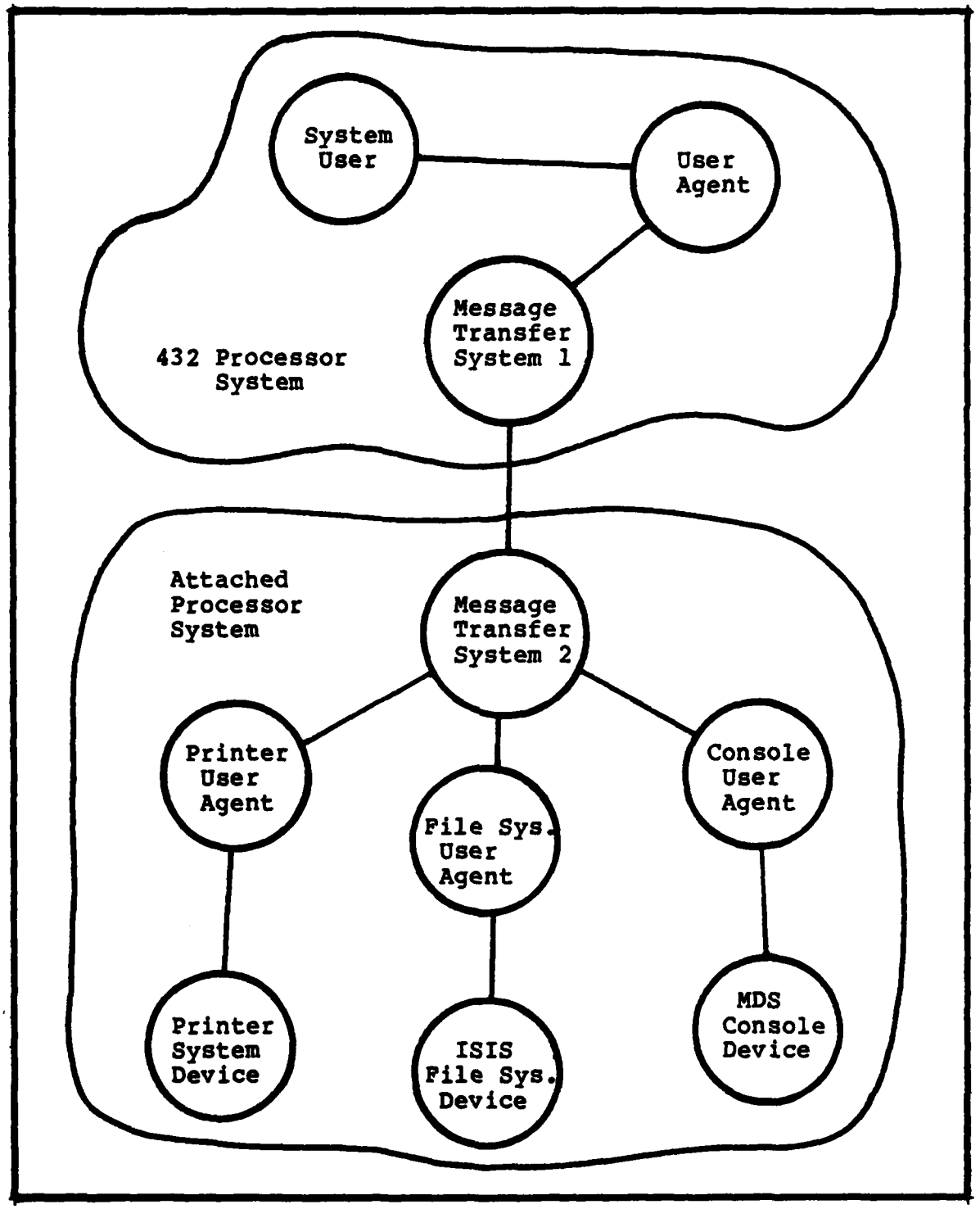


Figure H-5. AFIT/ENG 432/670 I/O Interface System

Operation of the Interface

This section describes the operation of the I/O Interface. The following paragraphs will present the general topology of the system and discuss the user interaction with the interface:

The I/O Interface contains three basic parts; the application program (hereafter, referred to as the User), the User Agent, and the Message Transfer System. The User is, simply, the applications program which requires access to I/O devices. The User Agent (UA) is the element which defines the way in which the User can interface with the I/O devices. Finally, the Message Transfer System (MTS) is the system that moves the I/O commands and replies between the User Agents and Device Agents (User Agents for the I/O devices).

Figure H-5 shows the organization of Users, UAs, MTS, and devices in the AFIT/ENG 432/670 System. This view is overly complex. The User process only interacts with the User Agent and, therefore, the system shown in Figure H-6 is more appropriate. The User can access any I/O device in the system through the User Agent.

The User Agent has two major parts; the send section and the receive section. Each section may be tailored to the needs of the User. The User Agent also has two communication ports; a receive port, and a reply port. These ports are created during the system initialization.

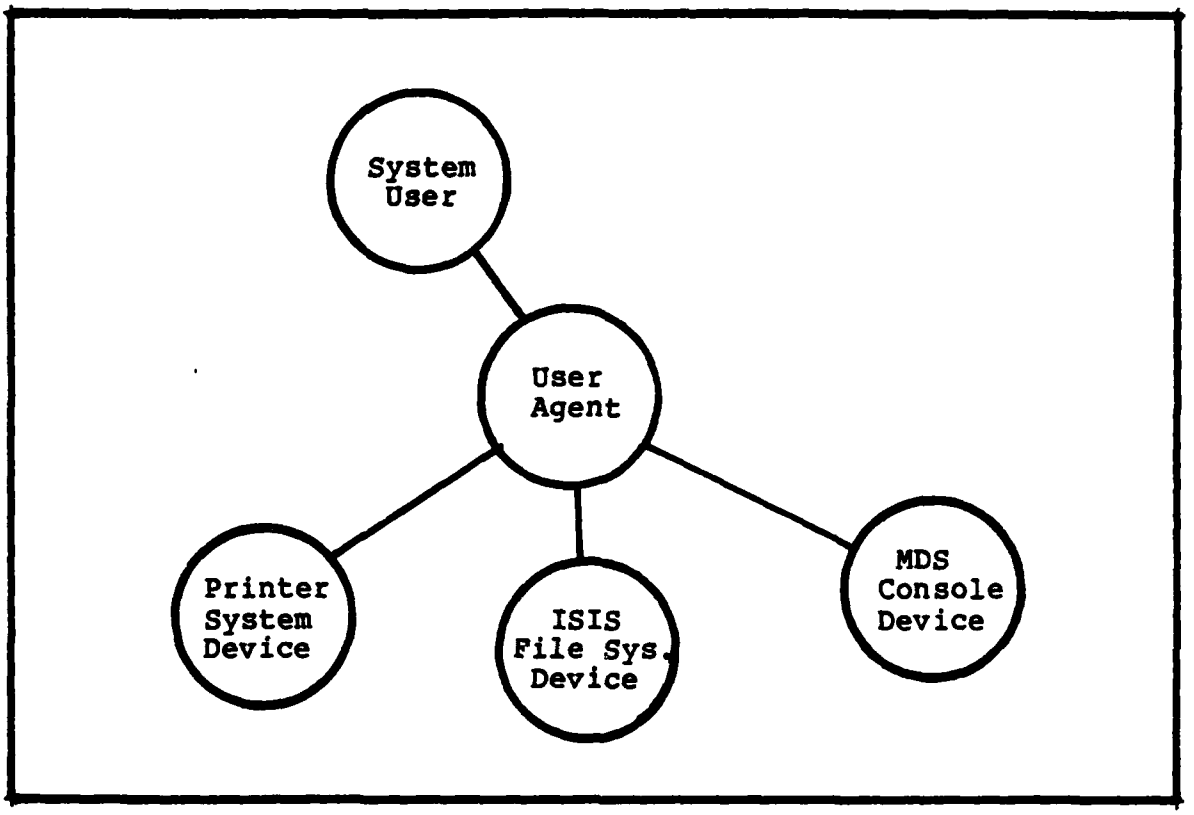


Figure H-6. User View of I/O Interface System

The reply port is used by the send section, which waits at that port for a response after sending a request to another User Agent. All request messages are sent, by the MTS, to the receive port.

The receive section is a static process that waits at the User Agent's receive port for a request message from another User Agent. The process then determines the I/O command and calls the User-supplied function appropriate for that request. The User function returns a status value which is then sent back to the requesting User Agent, as a reply message. Figure H-7 shows this process as a data

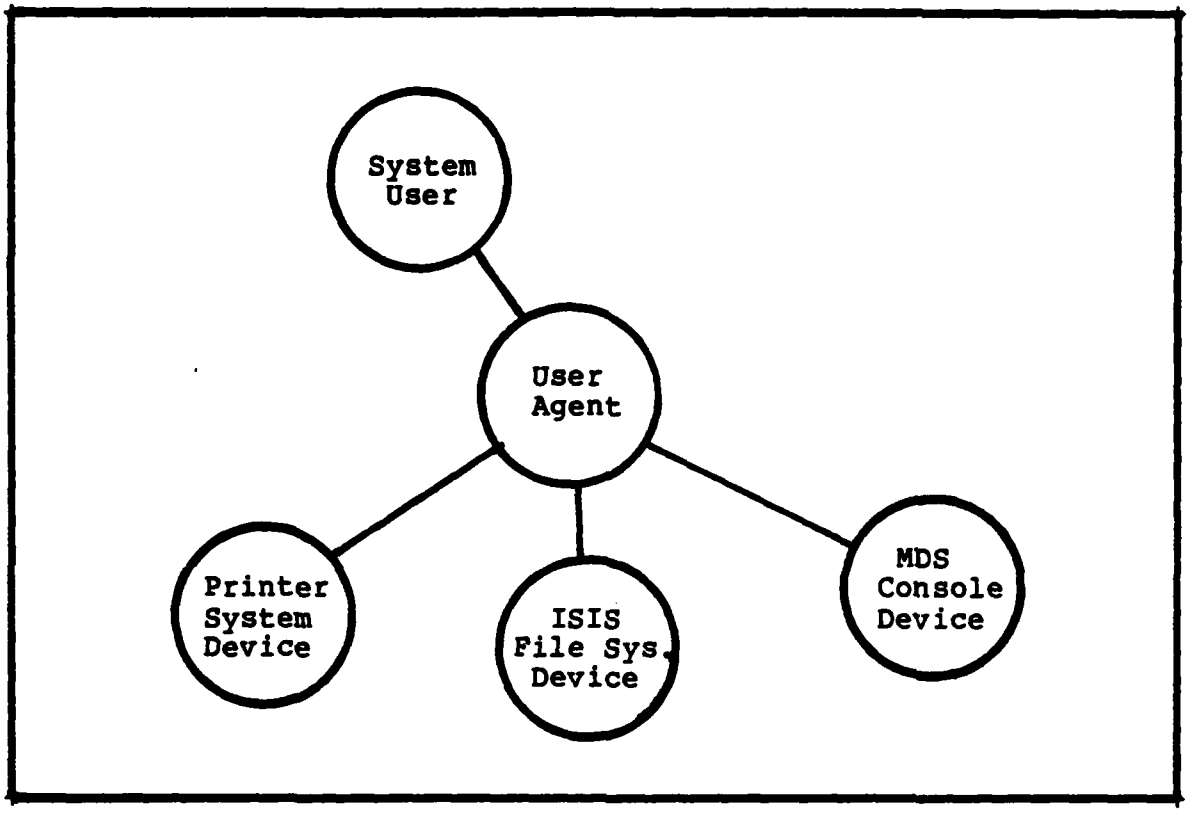


Figure H-6. User View of I/O Interface System

The reply port is used by the send section, which waits at that port for a response after sending a request to another User Agent. All request messages are sent, by the MTS, to the receive port.

The receive section is a static process that waits at the User Agent's receive port for a request message from another User Agent. The process then determines the I/O command and calls the User-supplied function appropriate for that request. The User function returns a status value which is then sent back to the requesting User Agent, as a reply message. Figure H-7 shows this process as a data

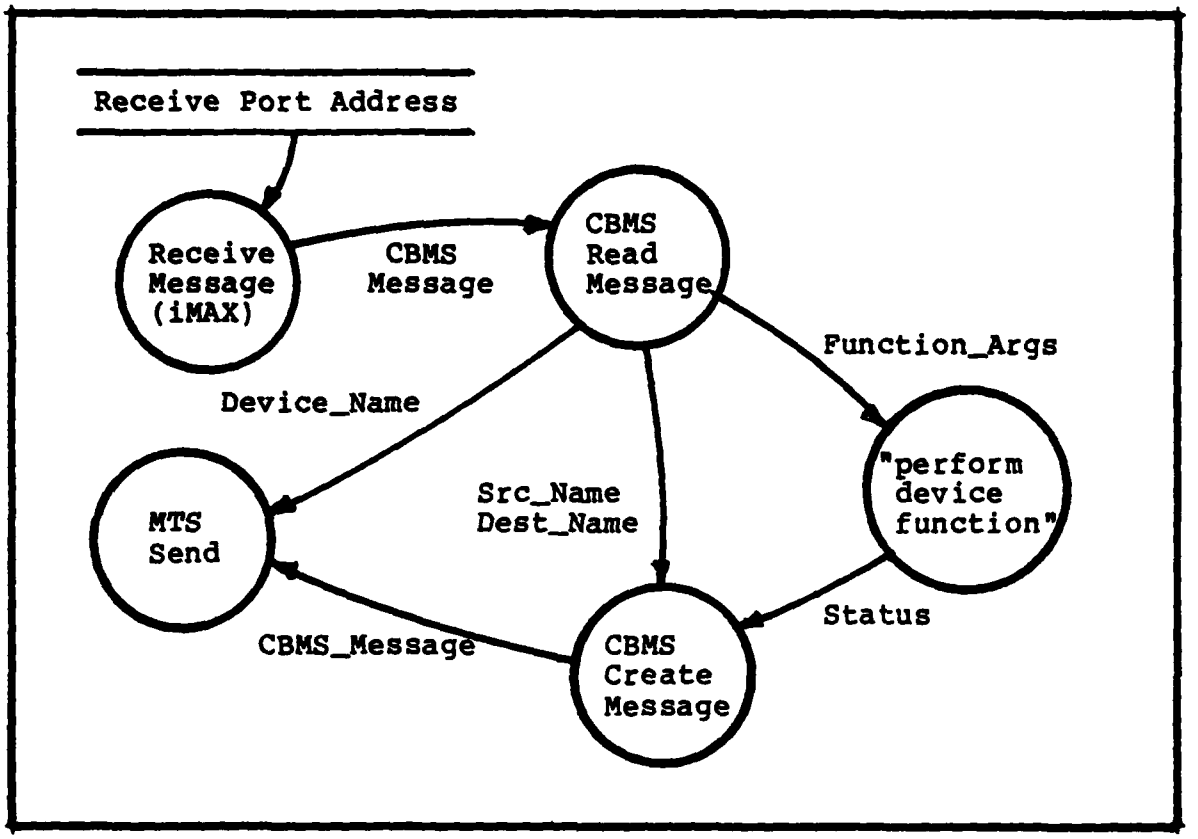


Figure H-7. User Agent Receive Process Data Flow

flow.

The receive section of the User Agent is written to work with a particular User. Each User is unique and performs a different set of functions in response to the I/O Interface commands. Whether the User is a peripheral device or a system process, the User Agent receive section must be tailored to call the functions of the User properly. The send section, however, may be the same for all users.

The send section, of the User Agent, contains a set of procedures which perform input and output functions using

TABLE H-V

I/O Interface Replies to Function Requests

I/O Interface Function	I/O Interface Reply Codes							
	0	1	2	3	4	5	6	7
	Ok	Invalid Command	End of File	Bad Data	- Device Error	- Closed	- Off	Busy
Open	x				x		x	x
Close	x				x		x	x
Read	x	x	x	x	x	x	x	x
Write	x			x	x	x	x	x
Page	x	x			x	x	x	x
Title	x	x		x	x	x	x	x
Delete	x	x			x		x	x
Rename	x	x		x	x		x	x
Reset	x	x			x		x	x
Get Config		x					x	
Set Config	x	x		x	x	x	x	x
Test	x				x		x	

the I/O Interface. Each procedure returns a reply code as an indication of the success or failure of the request. Table H-V shows the procedures of the send section and the reply codes they can return. The User Agent may contain all the procedures listed in the table (as in the case of the User Shell Agent, discussed later). However, to conserve memory, the send section only needs to contain the procedures that are required by that particular User.

The procedures of the send section generally perform the same steps. The send procedures use the facilities of the MTS to create a message containing the command and any data required. The message is sent to the User Agent for the device requested. That device's User Agent returns a

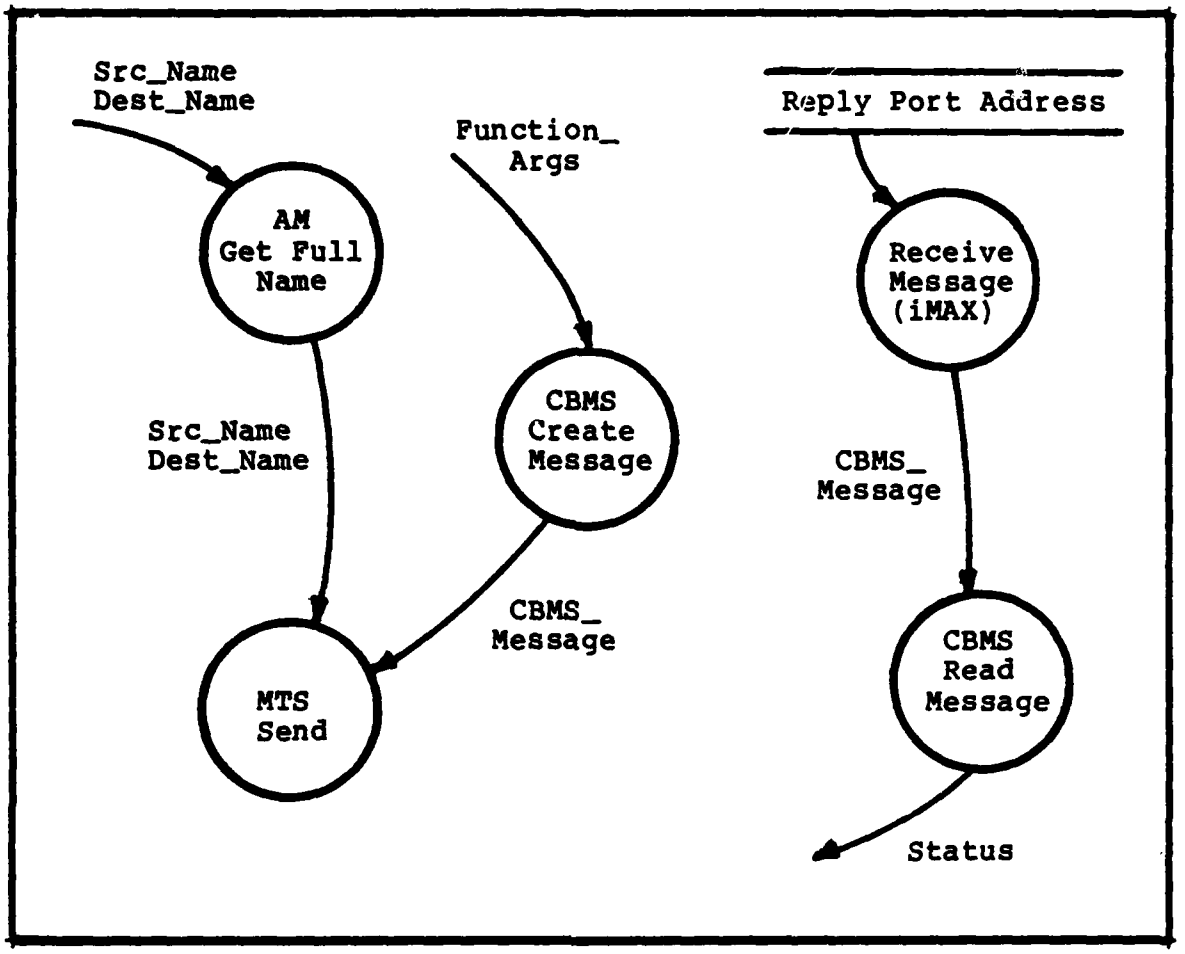


Figure H-8. Typical User Agent Send Procedure Data Flow

message containing the reply code that indicates the result of the operation. The send procedure, then, returns the status indication to the calling user process. Figure H-8 shows a typical send procedure as a data flow.

Each device has a User Agent similar to that described for the User. The devices that are currently implemented with the I/O Interface, on the AFIT/ENG 432/670 System, are described in the next section.

I/O Interface Devices

There are three I/O devices implemented in the current software of the I/O Interface. Each device is defined by two software packages; the device abstraction and the device agent.

The device abstraction package contains the functions which control the device and, thus, define the capabilities of the device for the system. For example, the Printer System package contains the procedures Ps_Open, Ps_Close, Ps_Print, Ps_Form_Feed, Ps_Title_Page, and Ps_Test. To the I/O Interface, the Printer System is a device which can perform only these functions. The implementations of these functions or the operation of the device driver routines which might be used to perform the functions are not important to the I/O Interface. The interface only requires knowledge of the arguments required by these functions and the status indications they provide in return.

The device agent package contains the procedures for mapping the interface commands to the device functions of the abstraction. The device agent may have two sections; one for sending interface commands to other devices and the other for receiving commands.

The receiving portion of the device agent is a process which accepts command messages from the interface and calls the necessary procedures of the device abstraction. Table H-VI shows the mapping of the I/O Interface functions onto

TABLE H-VI

Mapping of I/O Interface Commands to Device Functions

I/O Interface Command	Device Functions PS	IFS	S3C
Open	Open	Open	2
Close	Close	Close	2
Read	1	Read	Read
Write	Print	Write	Write
Page	Form Feed	2	Clear Screen
Title	Title Page	2	Clear Screen
Delete	1	Delete	1
Rename	1	Rename	1
Reset	Close	Reset	2
Test	Test	Test	Test
Get Config	1	1	1
Set Config	2	1	2

1 - Not Implemented, UA replies "command invalid"
2 - Not Used, UA replies "ok"

each device. When the device function is complete and returns a status value, the receive process calls procedures of the Message Transfer System to send a reply message to the requesting user of the I/O Interface. Thus, the receive process syncronizes the operation of the device with the command/reply sequence.

The send section of the device agent is only required when the device is capable of originating I/O Interface commands for other devices. None of the devices implemented for the Series III MDS have this capability. However, the send section of the package is simply a collection of

procedures, each of which implements an I/O Interface command, as discussed in the previous section.

Using the I/O Interface Devices

This section describes the operation of the current devices of the I/O Interface system. Each device has characteristics which must be known by the user for efficient operation. There are several characteristics that are common to all three devices:

-- The maximum buffer size (read or write) is 256 bytes in length.

-- The reply codes are 8-bit integers with values as given in Table H-VI, above.

-- The reply to the "test" command will be one of the following:

--- Ok - device ready to accept a command.

--- Device Busy - device is active and in use by another process.

--- Device Off - device did not receive the message or the User Agent did not respond to the request message.

The following sections describe the abstraction of each device and the operation of the procedures in the device software package:

Printer System Device. The Printer System is an implementation of a minimum number of procedures to provide printer services to the system. The functions are listed in Table H-VII. The functions define a device that is "output only" and can only be accessed by one user at a time.

TABLE H-VII

Printer System Functions

Function	Description
OPEN	open device to a user
CLOSE	close device to a user
PRINT	write a buffer to the printer
FORM_FEED	eject a page on the printer
TITLE_PAGE	print a title and eject a page
TEST	check printer status

The OPEN and CLOSE functions are used to ensure that the printer is allocated to only one user until that user is finished. The CLOSE function also performs the reset function, clearing any error condition in the device. The PRINT function allows the user to write a variable size buffer (0 to 256 bytes) of data to the printer. The FORM_FEED function requires no data and, simply, sends an ASCII form feed byte (0C hex) to the printer to eject a single page. The TITLE_PAGE function uses the data provided as a header, or title, and prints a formatted page, then moves to the top of the next page to prepare for printing again.

The software implementation of the printer assumes that the device is installed on SERIAL CHANNEL 2 of the Series III MDS. The ISIS operating system utilities are used to send the data to this port of the system. Obviously, if the device is not connected to this port, the system will not

TABLE H-VIII

ISIS File System Functions

Function	Description
OPEN	open file to a user
CLOSE	close file to a user
READ	read a buffer from an open file
WRITE	write a buffer to an open file
DELETE	delete a file from the disk
RENAME	rename a file on the disk
RESET	close all open files
TEST	check existence of a file

work. The TEST command will return a "Device Off" reply in this case. If the device is operating, the TEST procedure returns an indication of whether or not the printer has been opened by another user (and not yet closed).

The Printer System functions do not do any data transformations in this implementation. If necessary, the software could be modified to handle special requirements of the printer by modifying the data written to the device.

ISIS File System Device. The ISIS File System provides access to the ISIS operating system's disk storage facilities. The procedures implemented are listed in Table H-VIII. The procedures use ISIS operating system utilities to perform file operations and require valid ISIS format file names (Ref 3,4:2-1). All procedures of the ISIS File System require a valid file name as data in the message. Note that the use of these file operation does not require

any verification by the requesting process. All valid operations are performed when requested with no recourse.

The OPEN and CLOSE procedures are used to control the file access. There may be four files open, simultaneously, and each request message is checked to ensure that the requesting user is the same user that opened the file. The system will respond "Device Busy" to any other user requesting access to that file. If there are, already, four files open, then the system is busy to all other users.

The READ and WRITE procedures will only operate on files previously opened by that user. They each provide for buffer transfers of data (0 to 256 bytes) and the READ function returns an "end-of-file" indication when the last record of the file has been sent. A file that has been opened for a WRITE operation must be closed before a READ function will be accepted (likewise, for reading then writing).

The DELETE operation removes a file from the disk. The operation may not be requested on an open file. The system will indicate "device busy", if this is attempted. Also, the reply will indicate "invalid data" if the file does not exist on that disk. Generally, once a file has been deleted, it cannot be restored. Use of this function requires some caution.

The RENAME function required two valid file names and, like the DELETE function, cannot be performed on a file that

TABLE H-IX

Series III Console Device Functions

Function	Description
READ	read a buffer from keyboard
WRITE	write a buffer to the display
CLEAR_SCREEN	clear the console display
TEST	check console device

is open or does not exist. If the file names do not have identical drive numbers, the procedure indicates "bad data" and does not perform the function.

The RESET procedure, simply, closes all files opened by the requesting user. No other user's files are affected by this operation. No user may close another user's active files. This is the only command that does not require a file name as data. The RESET command always responds "Ok" if the system is operating.

The TEST command checks for the existence of the file given. If the file is not in the directory of the disk named, the procedure returns a "error" indication.

Series III Console Device. The Series III Console device is the implementation of a simple terminal source and sink system. There is no device allocation procedure for this device (i.e., OPEN and CLOSE procedures). Thus, input from several users can become intermixed on the display. Also, extended output from the keyboard may be sent to

different users, as well. The available procedures are listed in Table H-IX. As with the other devices, the maximum buffer size is 256 bytes.

The READ function accepts input from the Series III MDS console keyboard. The end of the input is signaled by entering a <RETURN>. The WRITE function displays the information in the message on the console display. The data will begin at the current cursor location. The eighth bit of the data characters is not changed by the procedure. Thus, any special control characters will be sent to the display without modification.

The CLEAR_SCREEN procedure requires no input. It, simply, writes an escape sequence to the console display (the two character sequence is 1B45 hex, the "Clear Display" code for the Heath H-19 terminal). This code may need to be modified, if the console terminal is changed, in the future.

The TEST function, for this device, does not actually check the device. Only the User Agent package and the existence of the device abstraction software is verified by this function. The procedure always returns "Ok" when called.

Adding New Devices

Future users of the AFIT/ENG 432/670 Computer System may wish to implement additional devices within the I/O Interface system. This IT/ENG 432/670 Computer System may wish to implement additional devices within the I/O

TABLE H-X

Guidelines for Adding New Devices to the I/O Interface

1. Define the device abstraction.
2. Define the command mapping onto the device abstraction.
3. Implement all I/O Interface commands in the device agent.
4. Provide procedures for device and device agent initialization.
5. Thoroughly test the procedures

Interface system. This section is intended to provide guidelines for adding new devices. In addition, these guidelines may be applied when modifying the existing device software packages.

There are two software packages that must be created for each device. The device abstraction package is the set of procedures which define the possible actions of the device. The other package is the device agent, the User Agent for the device, which defines the mapping of the I/O Interface commands onto the device functions and provides the device with access to procedures which can create I/O Interface messages.

In general, the best approach is to first examine the software of an existing device with similar functions. It may be possible to, simply, duplicate the procedures of the

existing packages. In any case, the structure of the software will provide an example of procedures which use the I/O Interface facilities.

If no existing software satisfies the requirements, then an organized approach may be taken to implement new devices. There are six steps (see Table H-X) which form an outline of a device development process.

First, create the abstraction. Determine the functions of the device and program those functions. The inputs and outputs should be clearly defined. Refer to Table H-V for the proper use of the reply codes to indicate the status of the function.

Second, define the mapping of the I/O Interface commands to the functions of the device. If a device function has no corresponding I/O command, it cannot be used by the system. Ensure that the arguments of the device function are properly retreived from the message format. Refer to Appendix D of the I/O Interface thesis for details of the message structure. Implement the mapping as function calls from the receive procedure of a new device agent package.

Next, ensure that all I/O Interface commands are handled by the device agent receive process. If there are any commands that do not apply to the new device, the device agent must return an appropriate reply code to the requesting user. Also, the initialization section of the

device agent should be implemented. This should be modeled from the software of an existing device. Any hardware initialization sequence should be included in this section.

The send section of the device agent should, then, be added. If the new device requires access to other system devices, the procedures which create and send I/O Interface messages should be included in the device agent. These procedures should be modeled after those of the User Shell Agent package implemented on the 432 processor system.

Finally, the device should be tested. If possible, the new software packages should be tested without other system devices active. The new software should be validated by thorough testing to ensure there is no undesired results. Stepwise, top-down, testing may be performed in the same sequence as followed in this design procedure.

Operation of the User Shell

The User Shell is a single user system interface, implemented as to demonstrate the use of the I/O Interface facilities. This section describes the configuration of the system and the operating commands available with the User Shell program.

Configuration. The User Shell provides for operator interface with the Intel 432/670 Computer System. The Debugger console is the operator station for command input to the Shell. This means that, during system operation, the

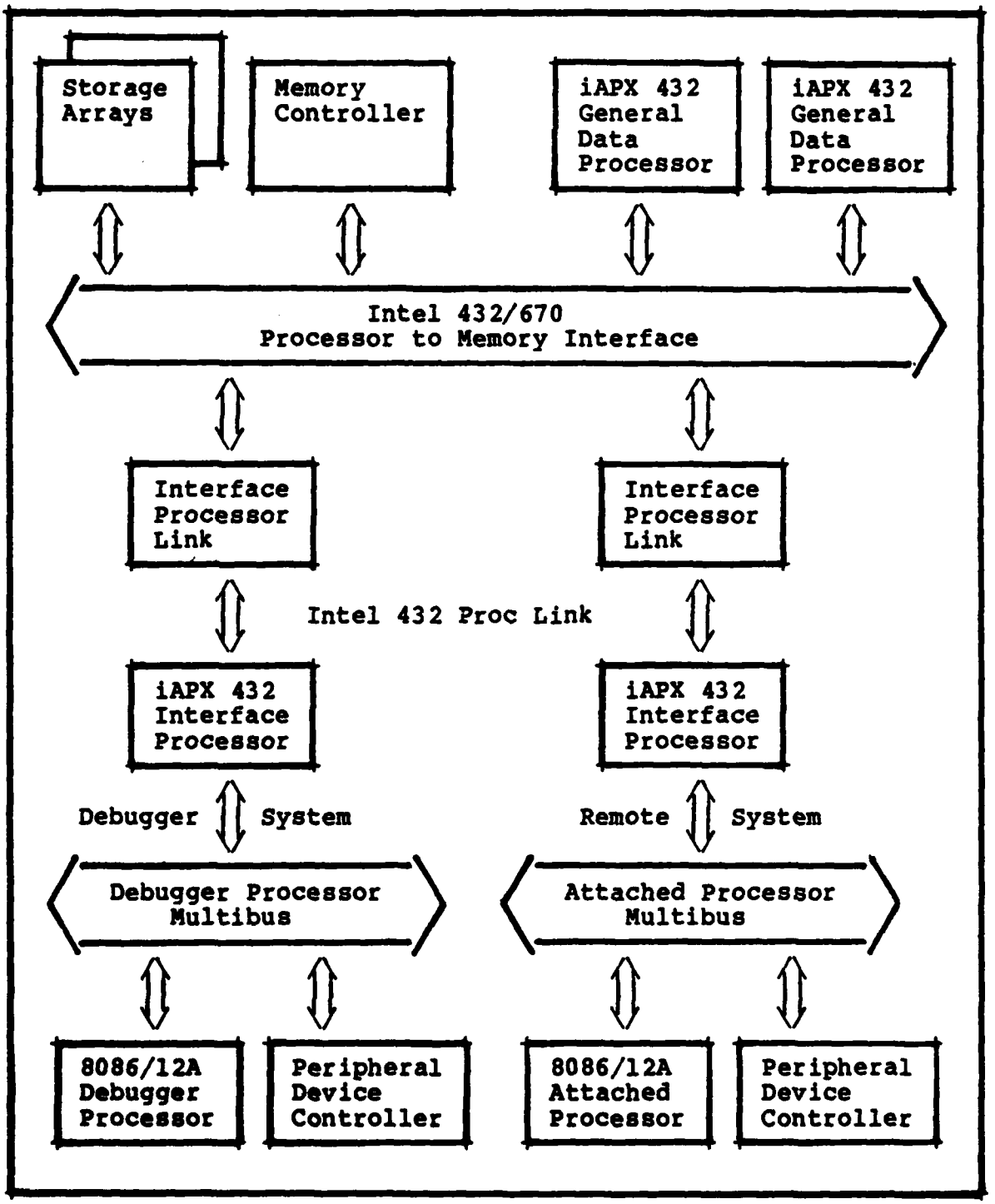


Figure H-9. User Shell Operating Hardware Configuration

TABLE H-XI

User Shell System Files

File Name	File Contents
Usa.mss	User Shell Agent Specification
Usa.mbs	User Shell Agent Body
User.mss	User Interface Specification
User.mbs	User Interface Body
Syscmd.mss	System Commands Specification
Syscmd.mbs	System Commands Body
Shell.mss	User Shell Specification
Shell.mbs	User Shell Body

Debugger system must remain as an Attached Processor to the 432/670 system. Figure H-9 shows this arrangement.

After the 432 Processor program is loaded and started, the Debugger workstation must be set to a mode which allows I/O with the 432. There are three commands which control the Debugger I/O mode:

CONTROL-C Place the debugger in Debugging Only mode

CONTROL-O Place the debugger in I/O Only mode

CONTROL-B Place the debugger in Debugging + I/O mode

A complete description of the Debugger commands, and their functions, can be found in the Intel 432 CDS Workstation Users Guide (Ref 14).

The software of the User Shell must be compiled and linked with the I/O Interface packages, using the facilities of the 432 CDS on the VAX-11/780. The Ada source files are listed in Table H-XI. Note that the User Shell Agent is

included for use with the User Shell.

This file must be used with the User Shell system because it provides access to the I/O Interface functions. If another User Agent package is being used with the 432 system software, then insure that there is no conflict with file names or package specifications. Generally, more than one process cannot use the same agent package.

It is also important to modify the process initialization package body (Pserp.mbs) to include the main procedure of the User Shell package (Shell.main). If this is not done, the User Shell process will not execute.

The Debugger console I/O facilities are very limited. There is no queueing of messages to be written to the console by the 432 processor. If two processes on the 432 attempt to write to the Debugger at the same time, one will write and the other process will be halted until the first process's I/O function is complete. Generally, this will cause the two processes to intermix their text on the Debugger console. The important corollary to this feature is that while the User Shell is waiting for a command from the operator, no other process can read or write at the Debugger console.

Shell Commands. The User Shell commands are provided to allow user interaction with the system while demonstrating the operation of the I/O Interface. The User Shell will prompt the operator for a command by

TABLE H-XII

Logical Operands for Command Syntax Definition
(Ref 24:163)

Symbol	Meaning
=	"is composed of"
+	"and"
[]	"choose one of (exclusive or)
<>	"at least one of (inclusive or)"
()	"optional"
{}	"iterations of"
**	"comment"
Delim	"one or more blanks"

writing the "prompt character" to the Debugger system console. The prompt character is ">". The command syntax is the following:

Command-Name Argument-1 Argument-2 ... Argument-n

where Command-Name is the name of the command (or an allowed abbreviation). The number "n" of required arguments depends on the command used. The Command-Name and arguments are separated by one, or more, blanks. The maximum length of a command line input is 79 characters. The syntax of each command is shown in a figure using the logical operands listed in Table H-XII.

There are three commands implemented in the User Shell; Help, Set, and Copy. The HELP command is provided simply for convenience of the user. The SET command allows the user to abbreviate the device names used in commands. The COPY command uses the functions of the I/O Interface to

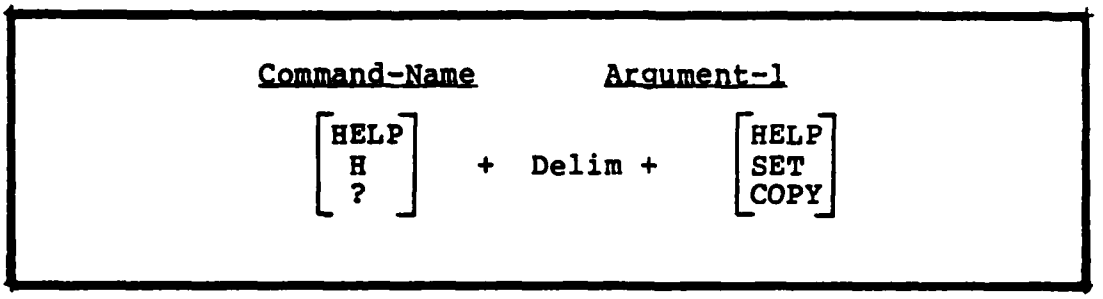


Figure H-10. Help Command Syntax

allow the user to transfer information from one device to another.

Help Command. The HELP command gives the user limited descriptions of the commands available and their syntax. There are three accepted Command-Name values for the HELP command; the full name, "HELP", the first letter, "H", or a question mark, "?".

There is only one argument allowed in the current implementation of the HELP command. Argument-1 may be the name of a system command; "HELP", "SET", or "COPY". If no argument is given after the Command-Name, then a list of the available commands is written to the console. Any incorrect value for Argument-1 will, also, get the list of commands. Figure H-10 summarizes the syntax of the HELP command.

The information provided for each valid HELP command is shown in Tables H-XIII through H-XVI. This information is stored as constant data in memory and modification requires rewriting the System_Commands package of the Ada source code. There are no references to this data from outside the

TABLE H-XIII

Help Command Response (Default)

COMMAND	HELP RESPONSE	
HELP	User Shell Commands:	HELP - Command Query SET - Default Naming COPY - Data Transfer

TABLE H-XIV

Help Command Response (Set Command Query)

COMMAND	HELP RESPONSE	
HELP SET	SET Command Use:	Default Device Naming Argument may be appended to the left side of abbreviated device name string
	SET Command Syntax:	Command Arguments SET DEFAULT Device-Name S
	Valid Device Name:	<Country>/<Network>/<Host>/<Device>
	where	<Country> must be RM67 <Network> must be NET0 <Host> is one of 432, MDS <Device> is one of [USR] on 432 [PTR, CON, or DSK] on MDS

TABLE H-XV

Help Command Response (Help Command Query)

COMMAND	HELP RESPONSE
HELP HELP	HELP Command Use: Command Query Response displays on Debugger console HELP Command Syntax: Command Argument HELP HELP H SET ? COPY

TABLE H-XVI

Help Command Response (Copy Command Query)

COMMAND	HELP RESPONSE
HELP COPY	COPY Command Use: Data Transfer from one system device to another COPY Command Syntax: Command Arguments COPY Device-Name-1 Device-Name-1 C Valid Device Name: <Country>/<Network>/<Host>/<Device> where <Country> must be RM67 <Network> must be NET0 <Host> is one of 432, MDS <Device> is one of [USR] on 432 [PTR, CON, or DSK] on MDS

Command-Name	Argument-1	Argument-2
[SET / S]	+ (Delim + DEFAULT + (Delim + "Device Name"))	

where

"Device Name" = (/) + "Element" + ({/ + "Element"})

Figure H-11. Set Command Syntax

package. Therefore, future implementations of this command software may use other data structures for this information storage.

Set Command. The SET command permits the user to set the value of a default device name string. Device names may, then, be entered as if the default string were added to the left side. The syntax for the SET command is shown in Figure H-11.

The format for the device name string is presented graphically in Figure H-12. A complete, or full, device name has four parts, separated by a slash (/); country, network, host, and device codes. When it is user, the file name follows the device name (e.g., "/DSK/filename.ext"). The format for file names (under ISIS operating system) is explained in the Intel publication Intellec Series III Microcomputer Development System Console Operating Instructions (Ref 4).

The SET command can make references, to the system

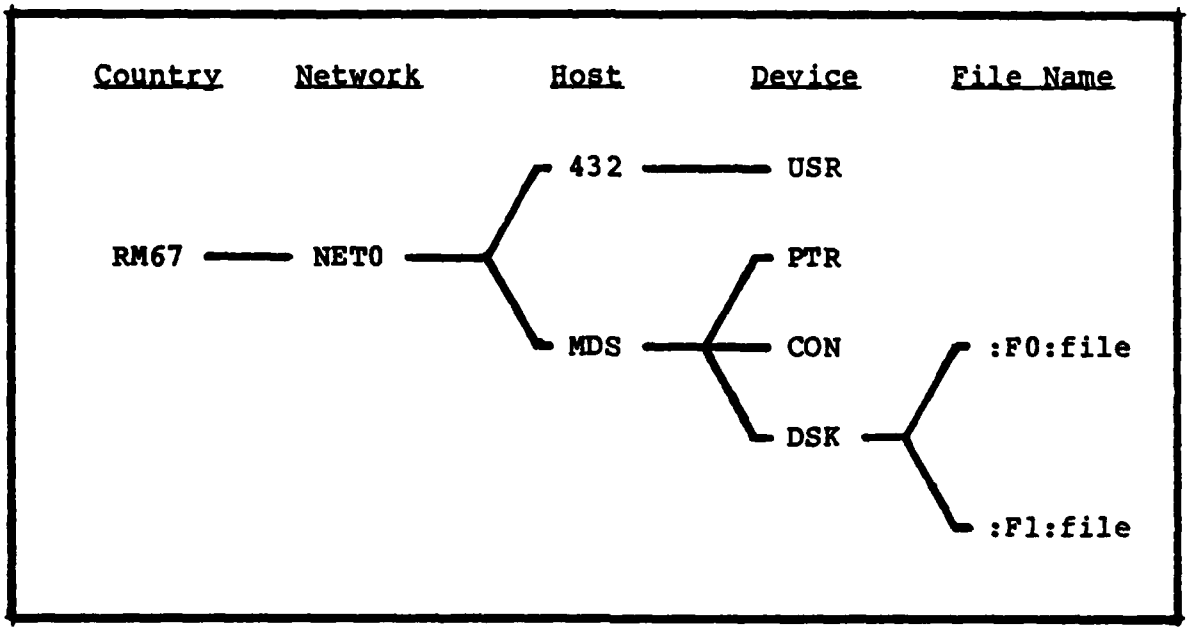


Figure H-12. I/O Interface Device Naming Structure

devices, much easier. For example, the full device name of the Printer System is "RM67/NET0/MDS/PTR". If the default name string is set to "RM67/NET0/MDS", then the Printer System may be referenced by the name "/PTR". The slash ("/") on the left indicates that the default should be added to the device name before the system reference is made.

Similarly, entering the this command:

```
SET DEFAULT RM67/NET0/MDS/DSK
```

would allow the following COPY command to be used:

```
COPY /:F0:OLDFILE.TXT  /:F1:NEWFILE.TXT
```

to copy one file to another. Without using the SET command, the COPY command line can become excessively long.

It is important to note that the SET command does <u>not</u> check for a valid device name when the default name string

```
Command-Name    Argument-1              Argument-2
[COPY]
[ C ] + (Delim +"Device Name" +(Delim +"Device Name"))
where
   "Device Name" = (/) + "Element" + ({/ + "Element"})
```

Figure H-13. Copy Command Syntax

is entered. This can result in invalid device name errors when using other system command.

Copy Command. The COPY command is used to move data from one device to another. The syntax for this command is shown in Figure H-13. The command accepts two arguments; the source and destination device names. If either of the arguments is missing, the console operator will be prompted to enter the missing element. Note that if only one device name is given in the command line, it is assumed to be the source device name and the operator will be asked for a destination name.

The COPY command performs a file transfer. That is, the system continues to transfer data, from the source to destination, until an end-of-file condition is found. Each system source device has a logical end-of-file indication to allow the device to function as a source for the COPY procedure.

When reading from a disk file (DSK device), an end-of-

file occurs when the end of the file data is reached. From either console device (USR or CON), use <CTRL>-Z to enter an end-of-file (i.e., press the "CTRL" key and the "Z" key simultaneously). The Printer System does not accept "read" commands, so, it has no need to generate an end-of-file.

Summary

The I/O Interface implementation on the AFIT/ENG 432/670 Computer System is an extensible and maintainable system. This manual has provided guidelines for using and maintaining the software system. Users should refer to the design thesis for the Interface (Ref 2), if more detail is required.

The attached bibliography includes the Intel publications that are necessary for operation of the Intel 432/670 Micromainframe Computer System. This documentation is extensive. It is hoped that this manual provides a concise presentation of the fundamental knowledge required to operate the I/O Interface system.

Bibliography

1. ANSI/MIL-STD-1815A. Ada Programming Language. Washington, D.C.: United States Government, Under Secretary of Defense, Research and Engineering, 1980.

2. Cole, Kenneth N. Design and Implementation of an Input/Output Interface Protocol for the Intel 432/670 Computer System, Unpublished MS Thesis. Wright-Patterson AFB, Ohio: School of Engineering, Air Force Institute of Technology, December 1983.

3. Intel Publication No. 121618-003. Intellec Series III Microcomputer Development System Programmer's Reference Manual. Santa Clara, California: Intel, Corp., 1981.

4. Intel Publication No. 121609-003. Intellec Series III Microcomputer Development System Console Operating Instructions. Santa Clara, California: Intel, Corp., 1981.

5. Intel Publication No. 142603-004. iRMX 80/88 Interactive Configuration Utility User's Guide. Santa Clara, California: Intel Corp., 1981.

6. Intel Publication No. 143232-002. iRMX 88 Reference Manual. Santa Clara, California: Intel Corp., 1981.

7. Intel Publication No. 143241-003. iRMX 88 Installation Instructions. Santa Clara, California: Intel Corp., 1981.

8. Intel Publication No. 171858-001 Rev. B. iAPX 432 Object Primer. Santa Clara, California: Intel Corp., 1981.

9. Intel Publication No. 171821-001. Introduction to the iAPX 432 Architecture. Santa Clara, California: Intel Corp., 1981.

10. Intel Publication No. 171867-001. Intel 432 System Summary: Manager's Perspective. Santa Clara, California: Intel Corp., 1981.

11. Intel Publication No. 171869-002. Reference Manual for the Ada Programming Language. Santa Clara, California: Intel Corp., 1981.

12. Intel Publication No. 171870-002. Intel 432 Cross Development System VAX Host User's Guide. Santa Clara, California: Intel Corp., 1982.

13. Intel Publication No. 171954-002. Introduction to the Intel 432 Cross Development System. Santa Clara, California: Intel Corp., 1982.

14. Intel Publication No. 172097-002. Intel 432 Cross Development System Workstation User's Guide. Santa Clara, California: Intel Corp., 1982.

15. Intel Publication No. 172098-002. System 432/600 System Reference Manual. Santa Clara, California: Intel Corp., 1982.

16. Intel Publication No. 172103-002. iMAX 432 Reference Manual. Santa Clara, California: Intel Corp., 1982.

17. Intel Publication No. 172174-001. Asynchronous Communication Link User's Guide. Santa Clara, California: Intel Corp., 1981.

18. Intel Publication No. 172283-001. Reference Manual for the Intel 432 Extensions to Ada. Santa Clara, California: Intel Corp., 1981.

19. Intel Publication No. 9800466,rev C. PL/M-86 Programming Manual for 8080/8085-Based Development Systems. Santa Clara, California: Intel, Corp., 1981.

20. Intel Publication No. 9800478,rev D. PL/M-86 Compiler Operating Instructions. Santa Clara, California: Intel, Corp., 1981.

21. McNamara, John E. Technical Aspects of Data Communication. Bedford, Massachusetts: Digital Equipment Corporation, 1977.

22. Phister, Paul W., Jr. Protocol Standards and Implementation within the Digital Engineering Laboratory Computer Network (DELNET) using the Universal Network Interface Device (UNID). Unpublished MS thesis. Wright-Patterson AFB, Ohio: School of Engineering, Air Force Institute of Technology, October 1983.

23. Smith, Lynn M. Intel 432/670 Computer System User's Guide. Unpublished text. Wright-Patterson AFB, Ohio: School of Engineering, Air Force Institute of Technology, June 1983.

24. Weinberg, Victor. Structured Analysis. New York, New York: Yourdon Press, 1979.

APPENDIX I

I/O Interface Data Flow Diagrams

This appendix contains the data flow diagrams for the I/O Interface. These diagrams have been extracted exactly as they appear in Chapter III and are collected here for reference. The full discussion of the logic development for these diagrams can be found in the main body of this report.

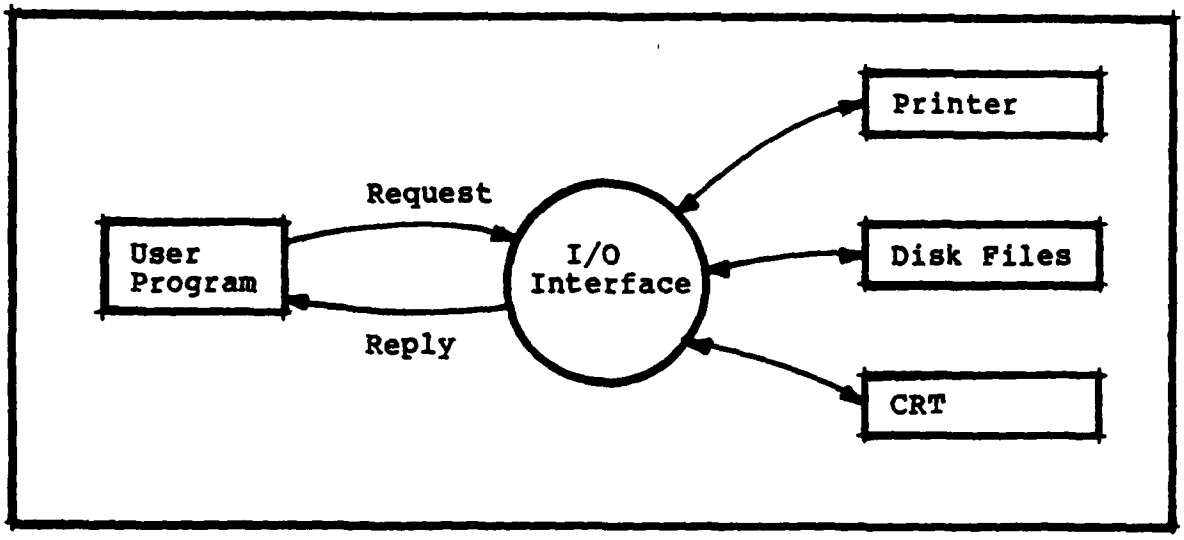


Figure 3-1. I/O Interface Context Diagram

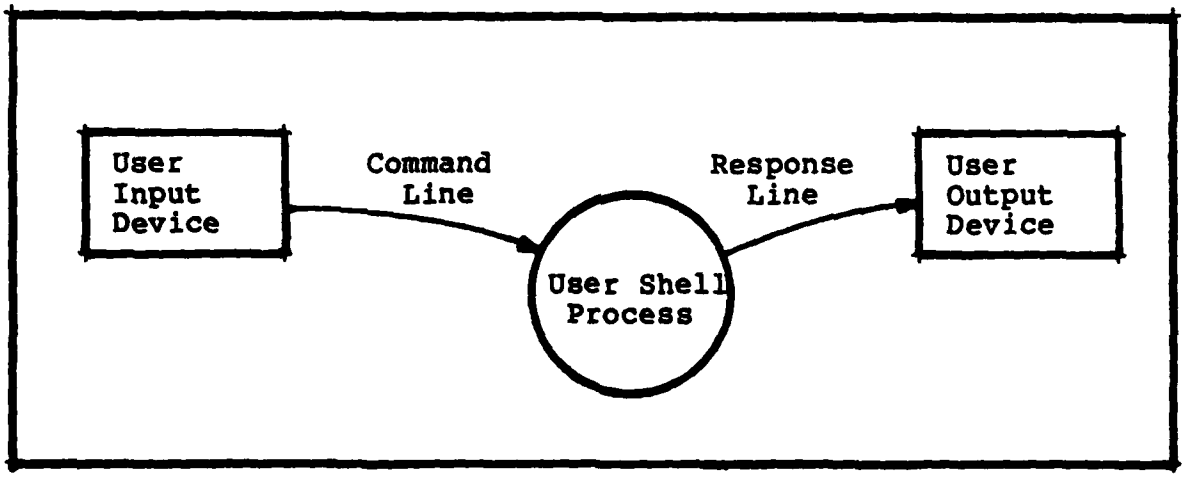


Figure 3-3. User Shell Context Diagram

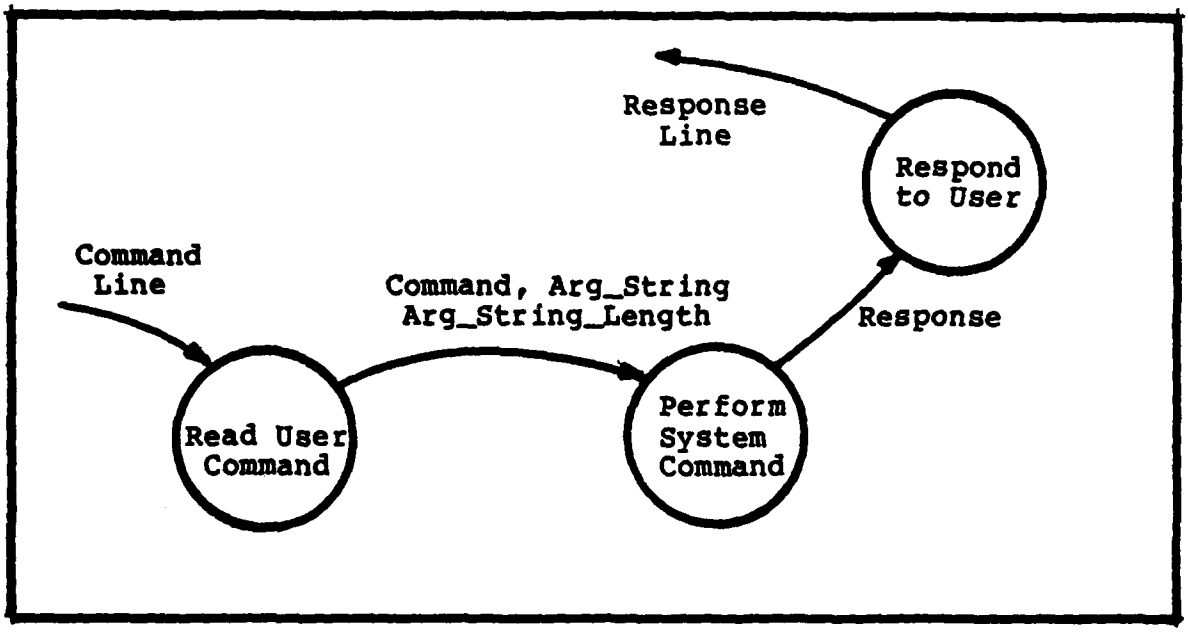


Figure 3-4. User Shell Main Program Data Flow

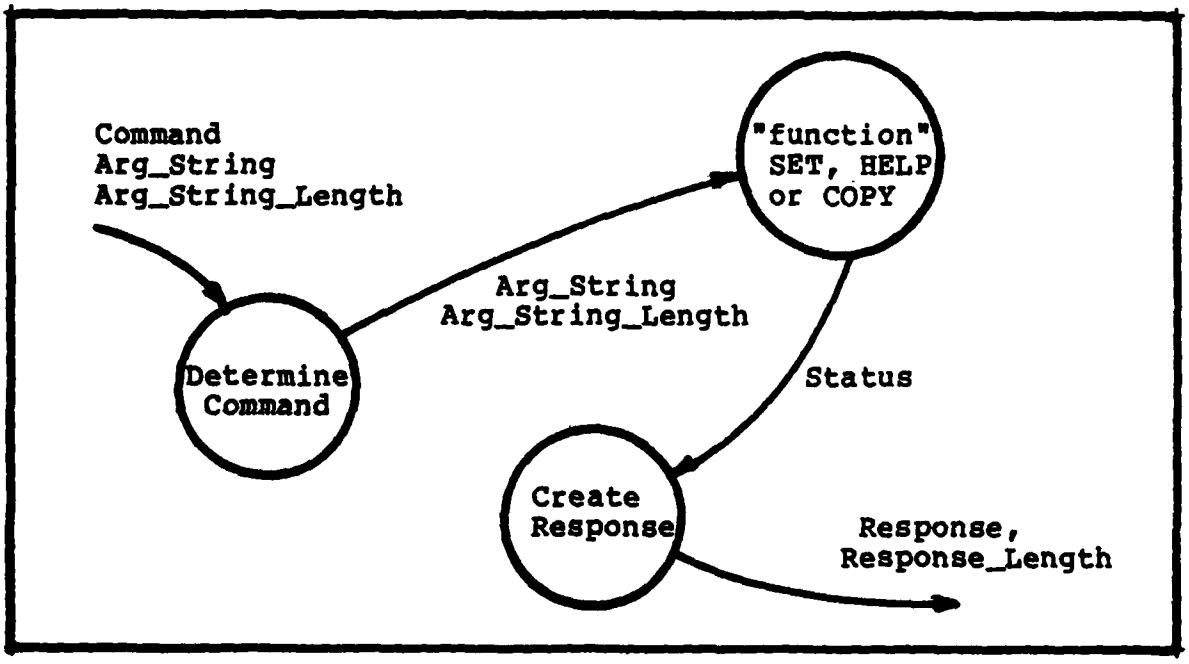


Figure 3-5. Perform_System_Command Data Flow

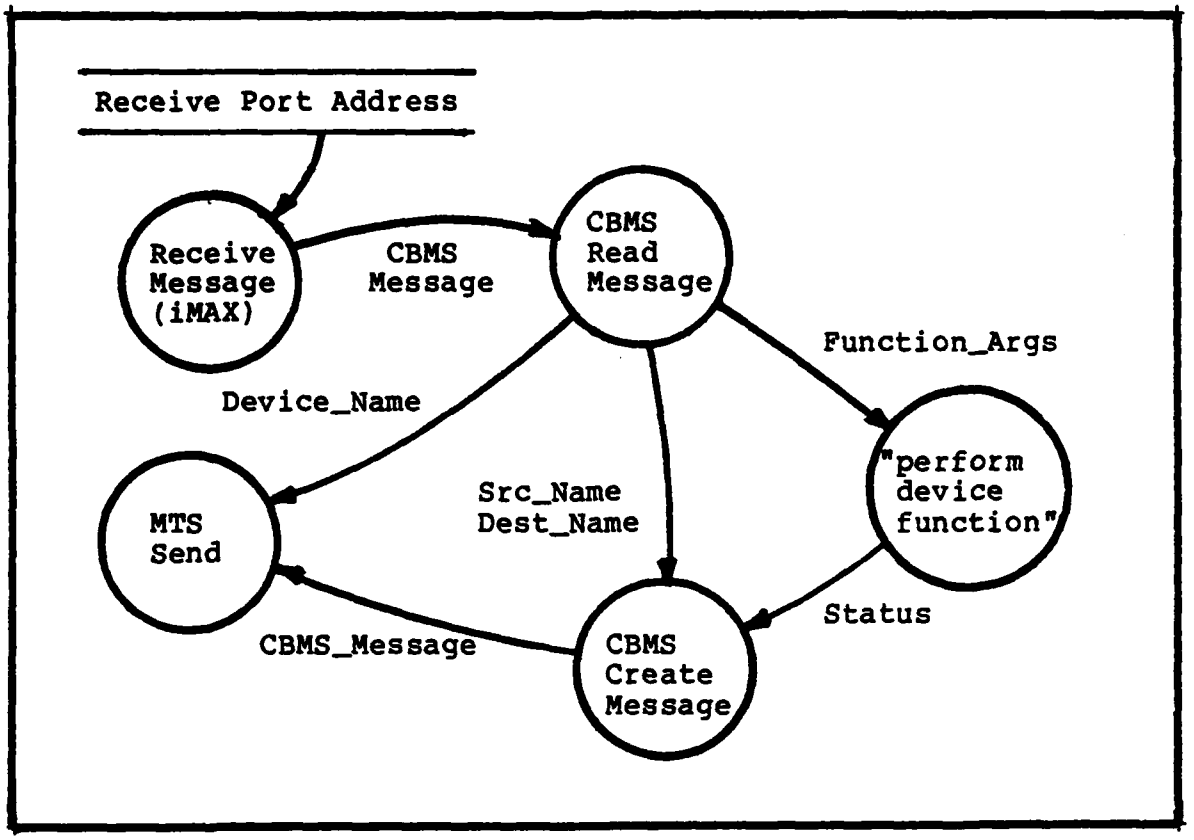


Figure 3-6. Typical User Agent Receive Procedure Data Flow

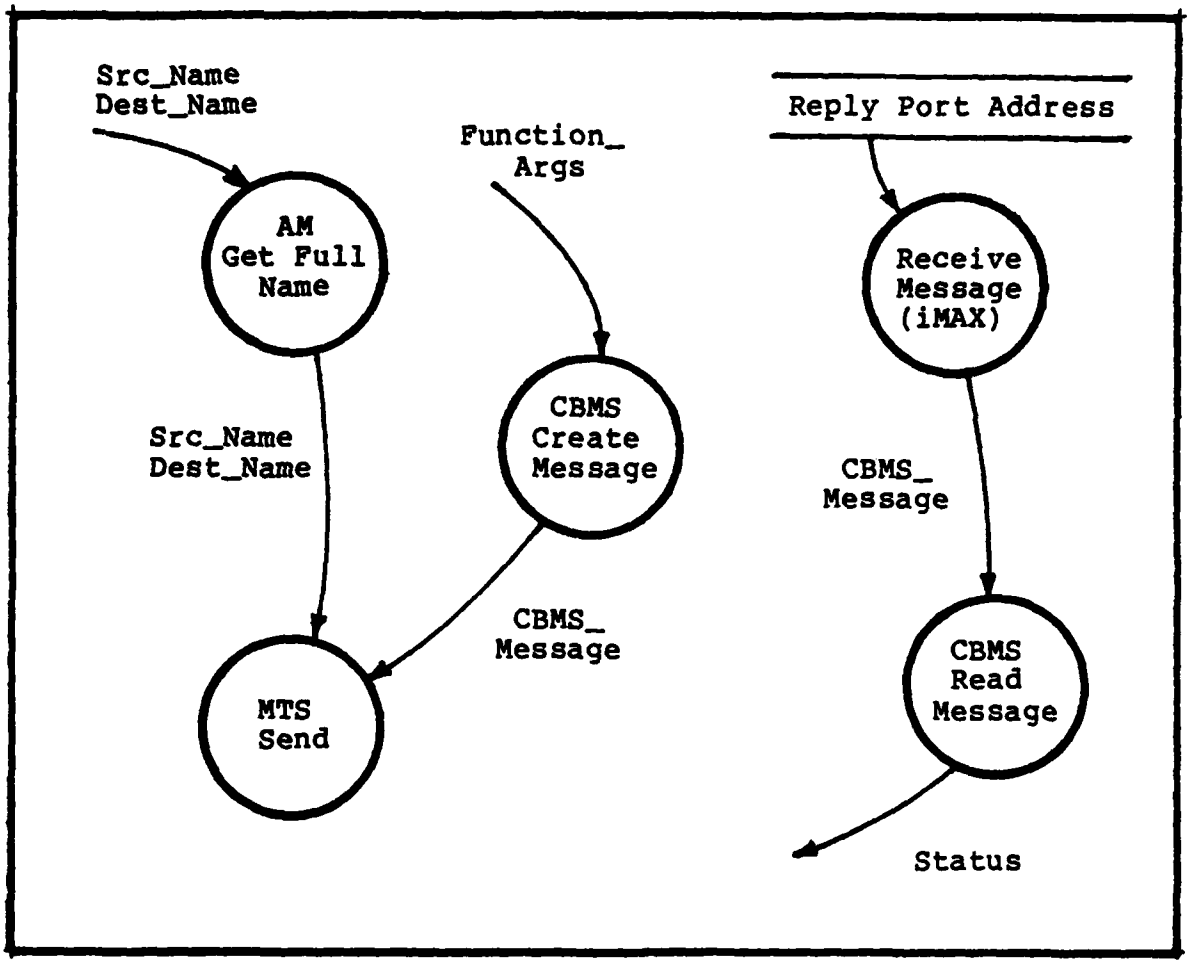


Figure 3-7. Typical User Agent Send Procedure Data Flow

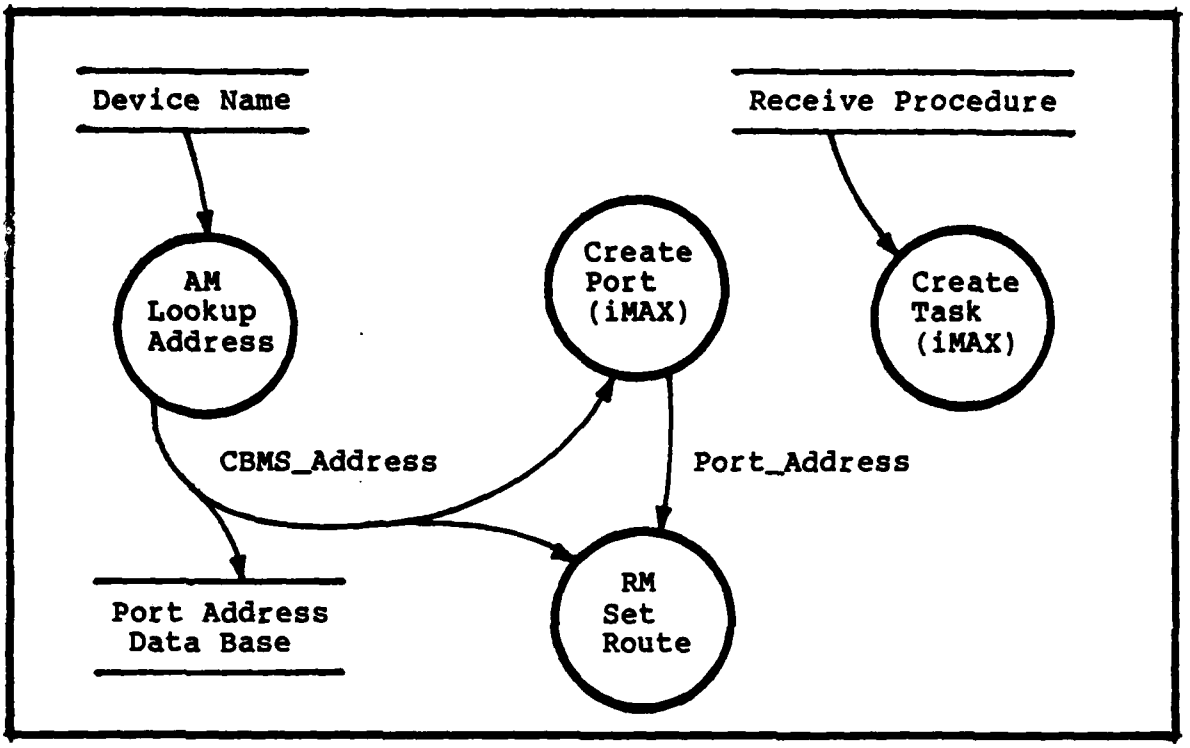


Figure 3-8. Typical User Agent Initialization Data Flow

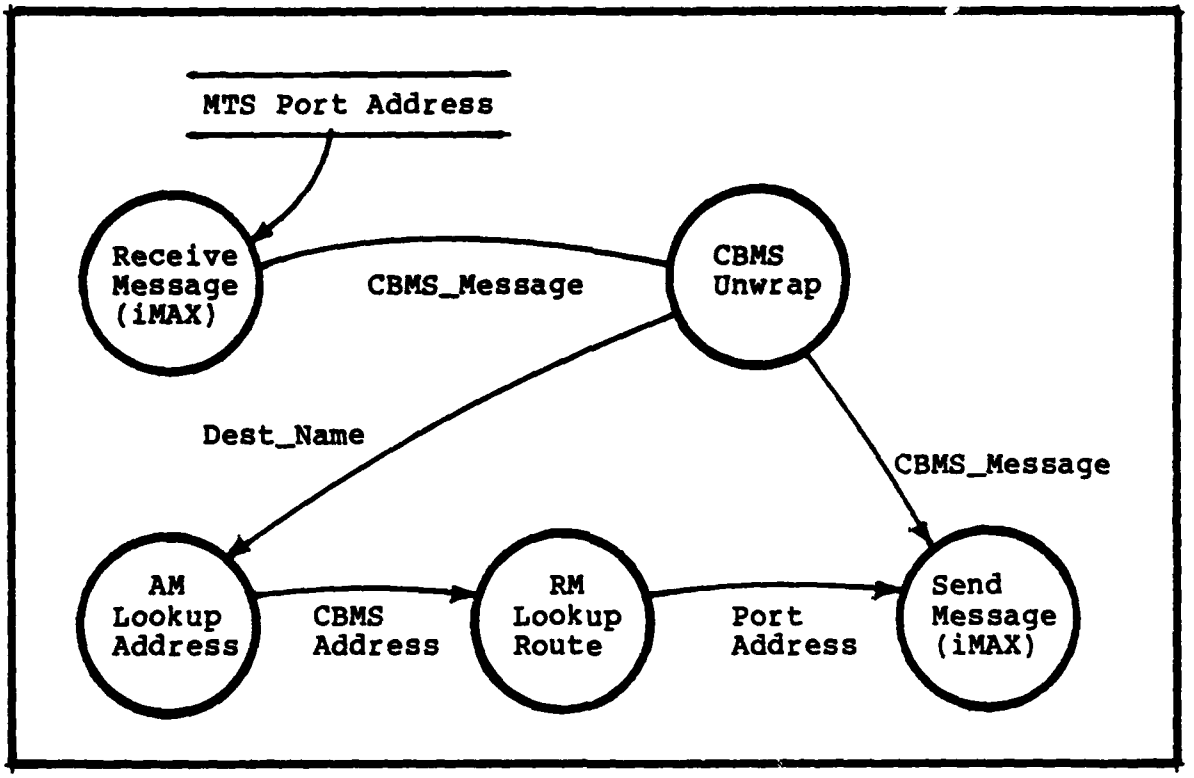


Figure 3-10. Mts_Receive Data Flow

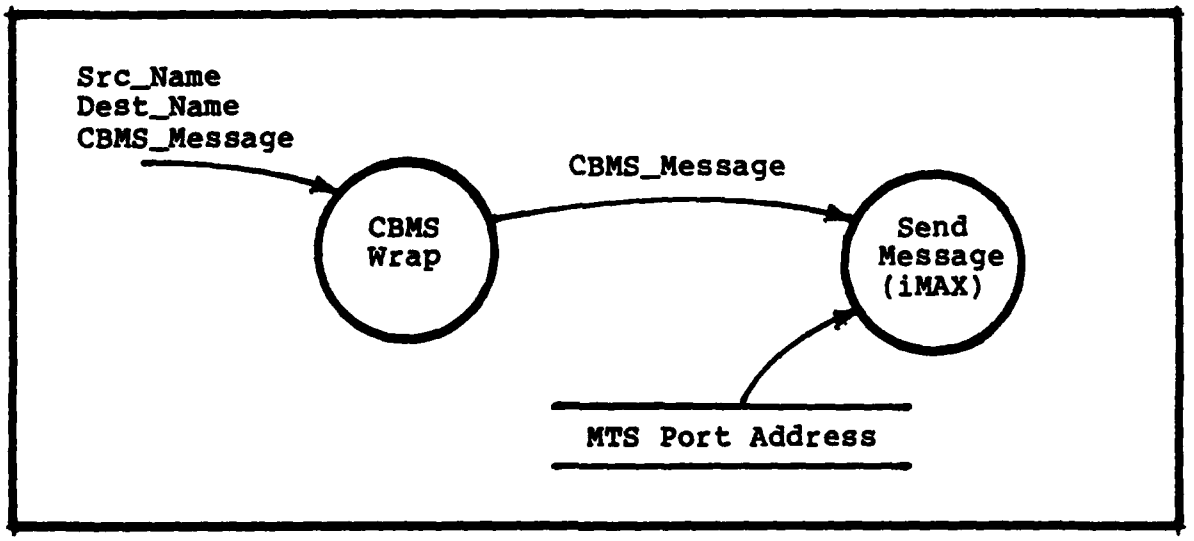


Figure 3-11. Mts_Send Data Flow

APPENDIX J

I/O Interface Software Structure Charts

This appendix contains the structure charts for the I/O Interface. These diagrams reflect the design of the I/O Interface software modules described in Chapter III. Code listings for these modules can be found in Volume II of this thesis.

Note that there are identical modules on the 432 Processor system and the Attached Processor system. In the case where the modules are named and numbered the same, the only difference is the language of the implementation program (see Appendix H). The data dictionary, included in Volume II, provides a cross reference for the module and variable names used in these charts.

In general, a single structure chart is sufficient to show the organization of each system. However, when there is a difference in the structure of the modules a separate chart is presented for each system. The following page contains a list of the charts provided.

I/O Interface Structure Charts

System/Module	Figure
432 System	
0.1 432 System Initialization	J-1
User Shell	
0.2 Main	J-3
System Commands	
2.1 Perform System Command	J-4
2.1.1 Determine Command	J-5
2.1.1.1 Set	J-6
2.1.1.2 Help	J-7
2.1.1.3 Copy	J-8
User Shell Agent	
6.1.1 USA_Receive	J-21
6.2 USA_Open	J-9
6.3 USA_Close	J-10
6.4 USA_Read	J-11
6.5 USA_Write	J-12
6.6 USA_Page	J-13
6.7 USA_Title	J-14
6.8 USA_Delete	J-15
6.9 USA_Rename	J-16
6.10 USA_Reset	J-17
6.11 USA_Get_Config	J-18
6.12 USA_Set_Config	J-19
6.13 USA_Test	J-20
Message Transfer System	
10.1.1 MTS_Receive	J-25
Attached Processor System	
0.1 Attached Processor Initialization	J-2
Printer System Agent	
7.1.1 PSA_Receive	J-22
ISIS File System Agent	
8.1.1 IFSA_Receive	J-23
Series III Console Agent	
9.1.1 S3CA_Receive	J-24
Message Transfer System	
10.1.1 MTS_Receive	J-25

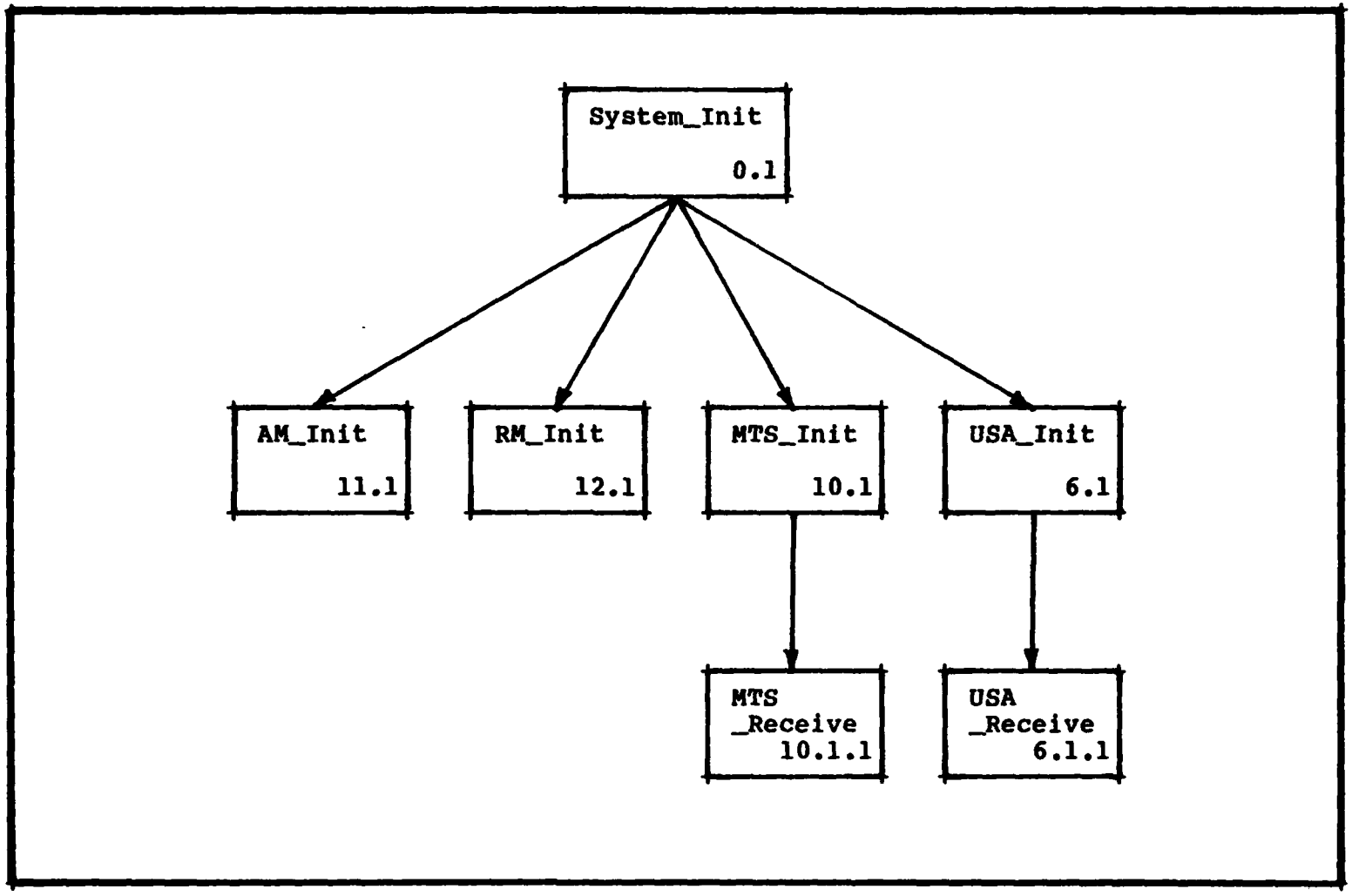


Figure J-1. 432 System Initialization Structure Chart (0.1)

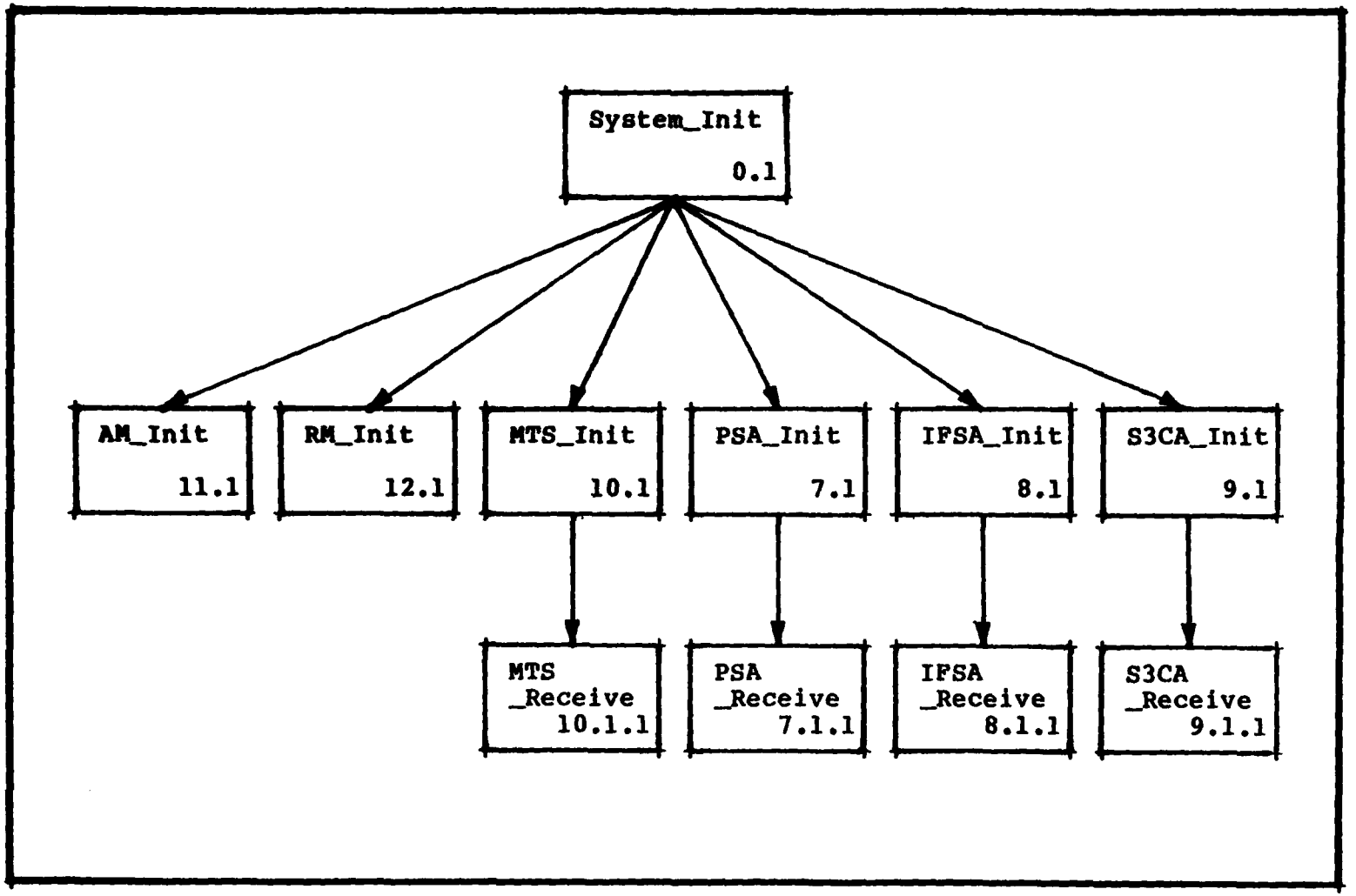


Figure J-2. Attached Processor System Initialization Structure Chart (0.1)

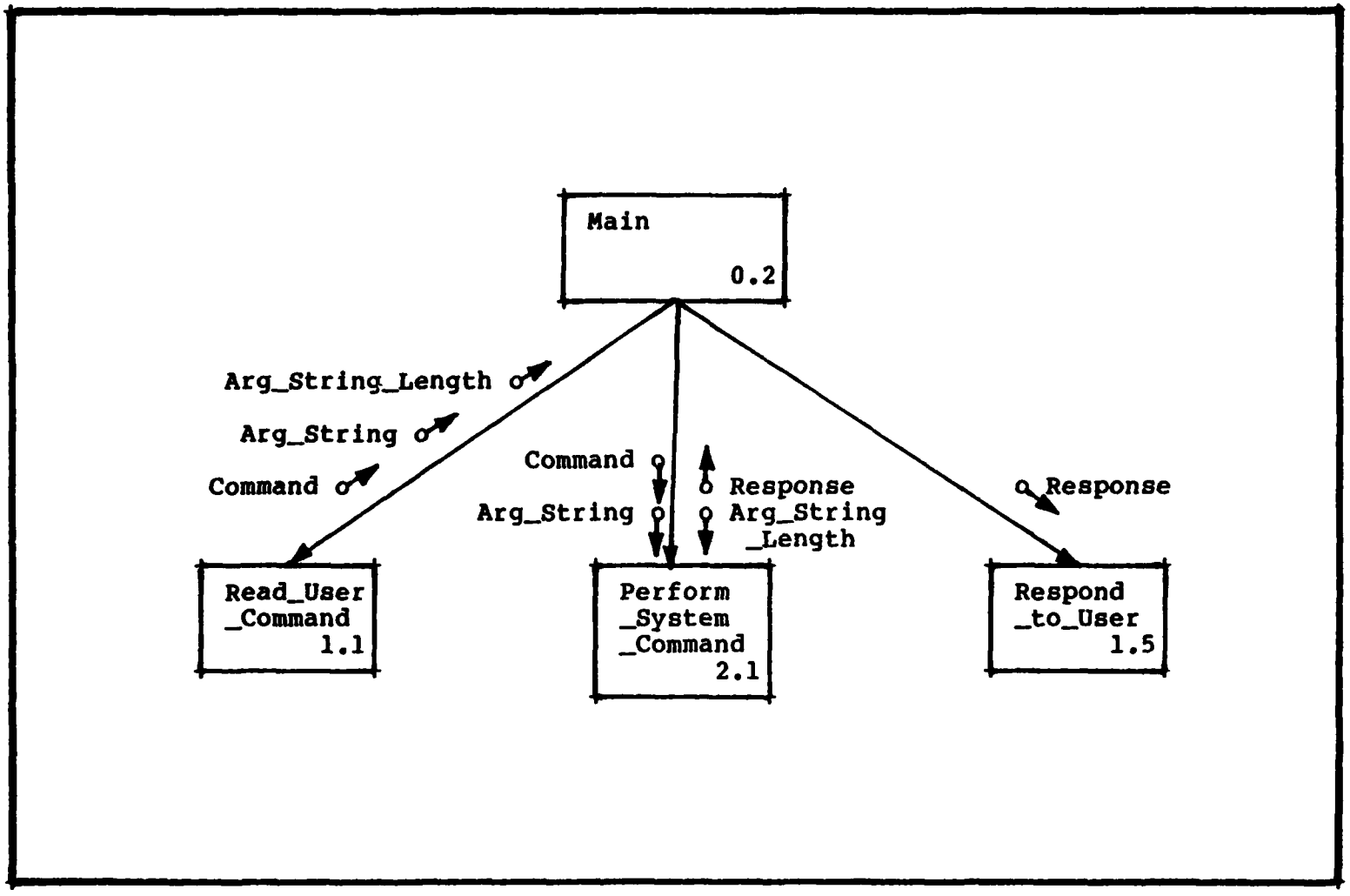


Figure J-3. Main (User Shell) Structure Chart (0.2)

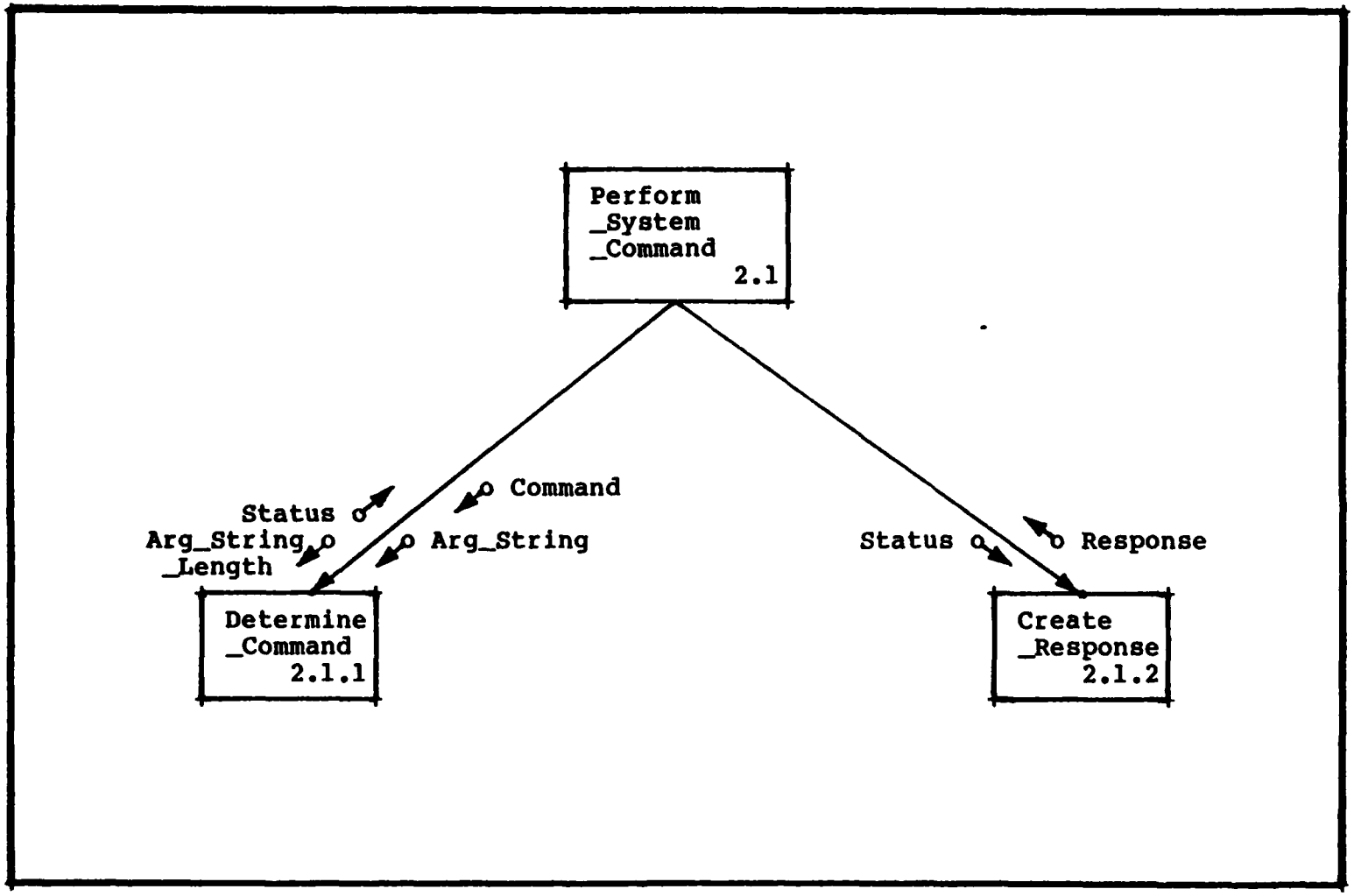


Figure J-4. Perform System Command (System Commands) Structure Chart (2.1)

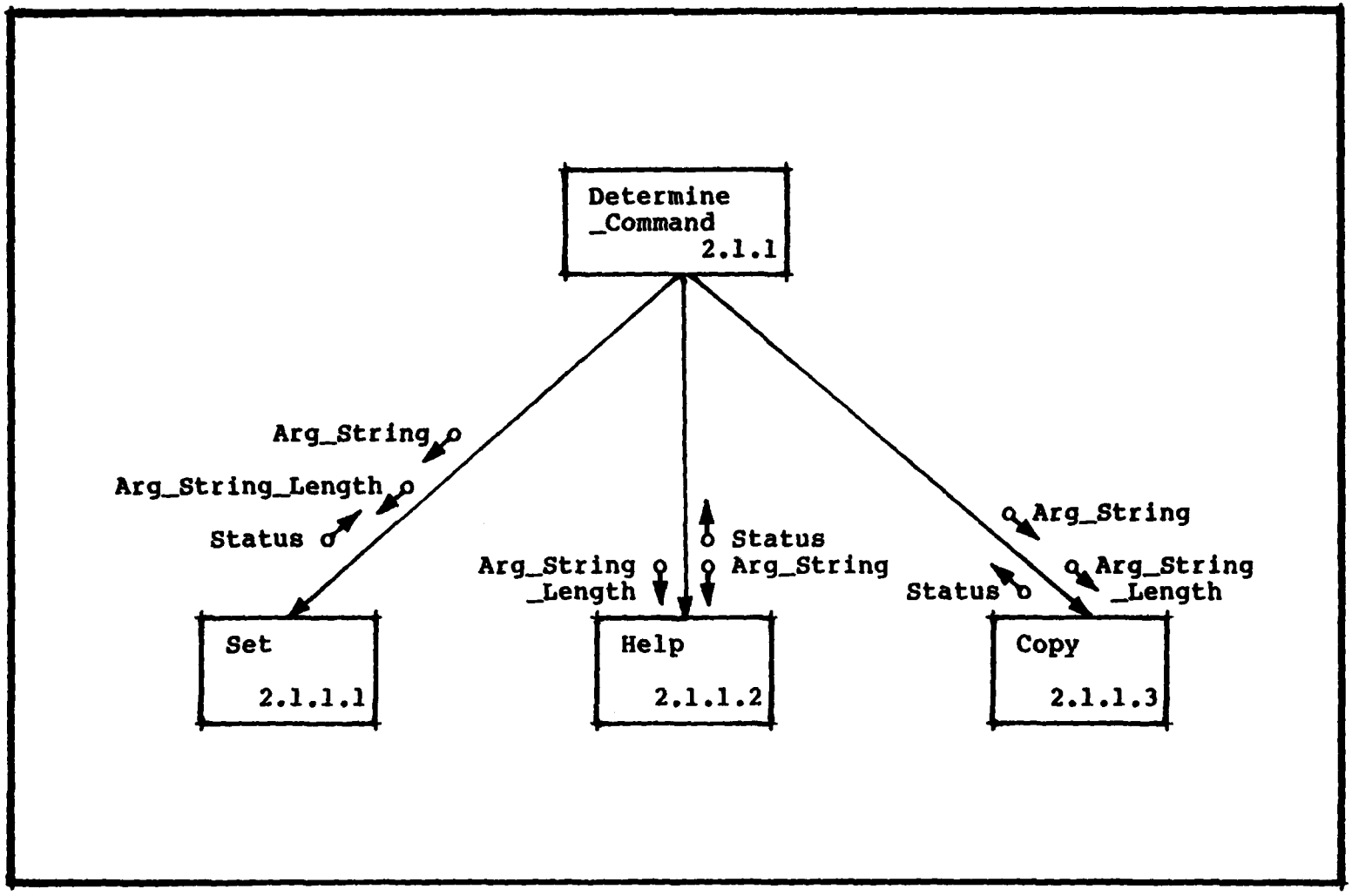


Figure J-5. Determine Command (System Commands) Structure Chart (2.1.1)

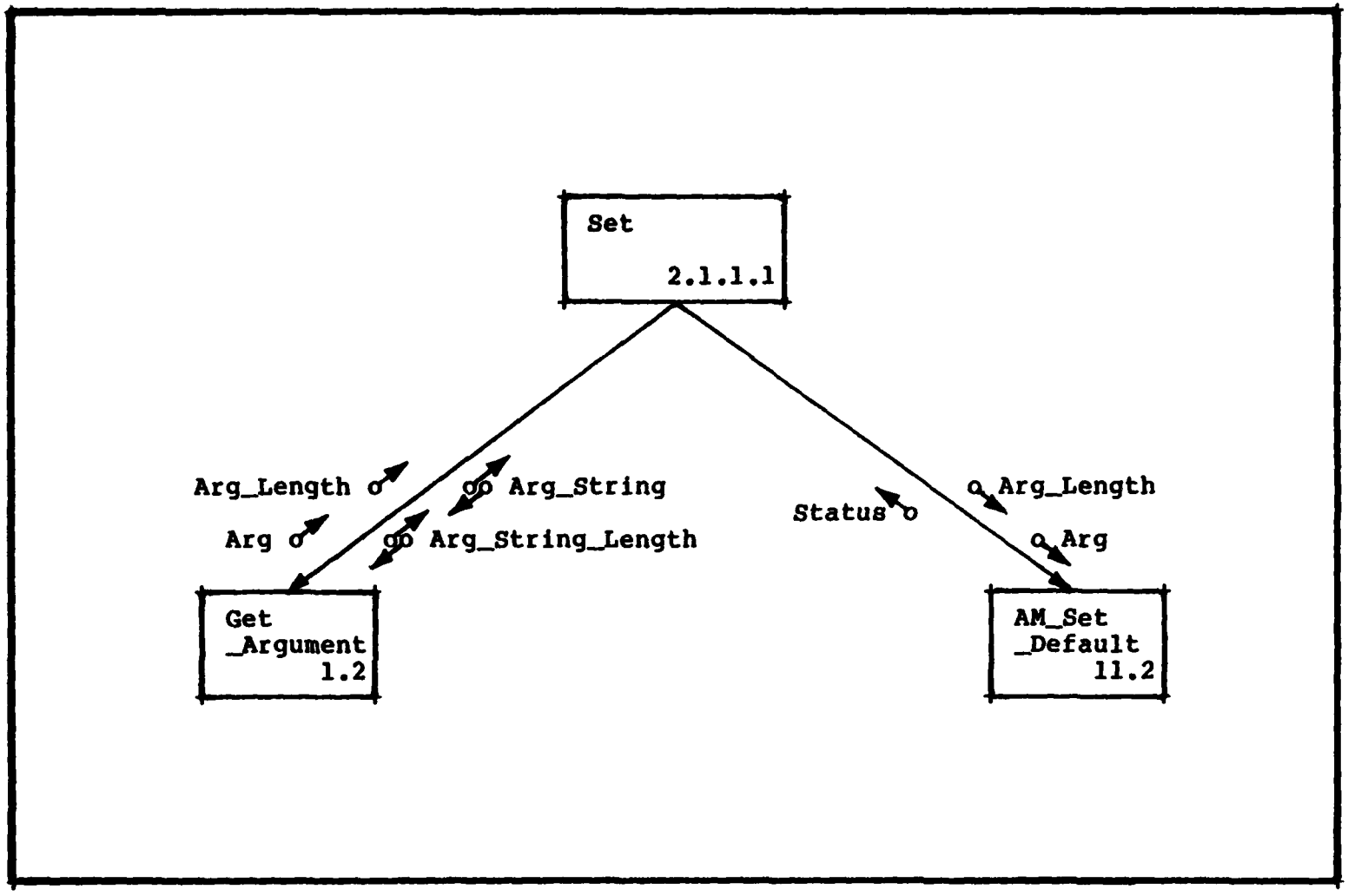


Figure J-6. Set (System Commands) Structure Chart (2.1.1.1)

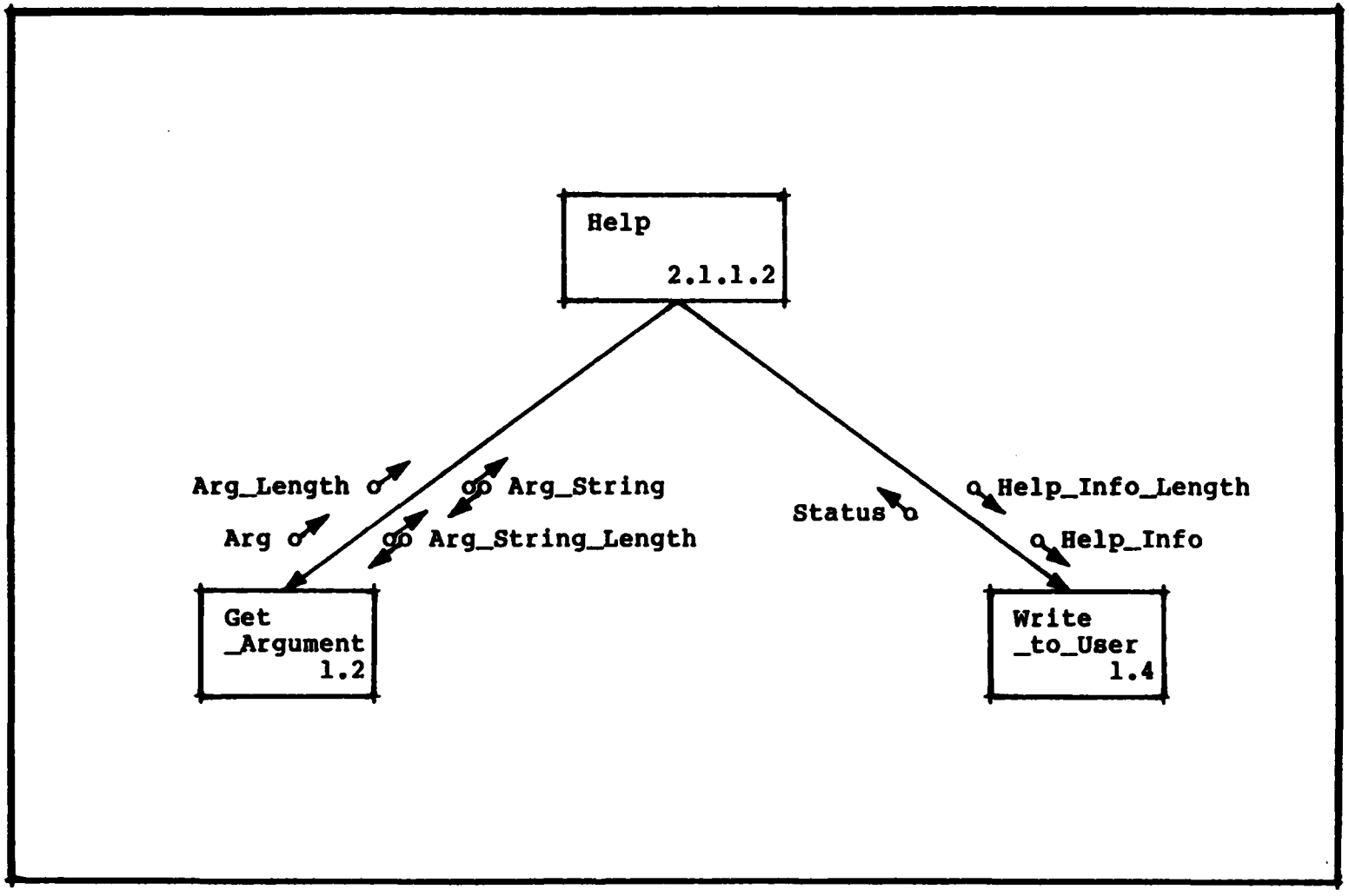


Figure J-7. Help (System Commands) Structure Chart (2.1.1.2)

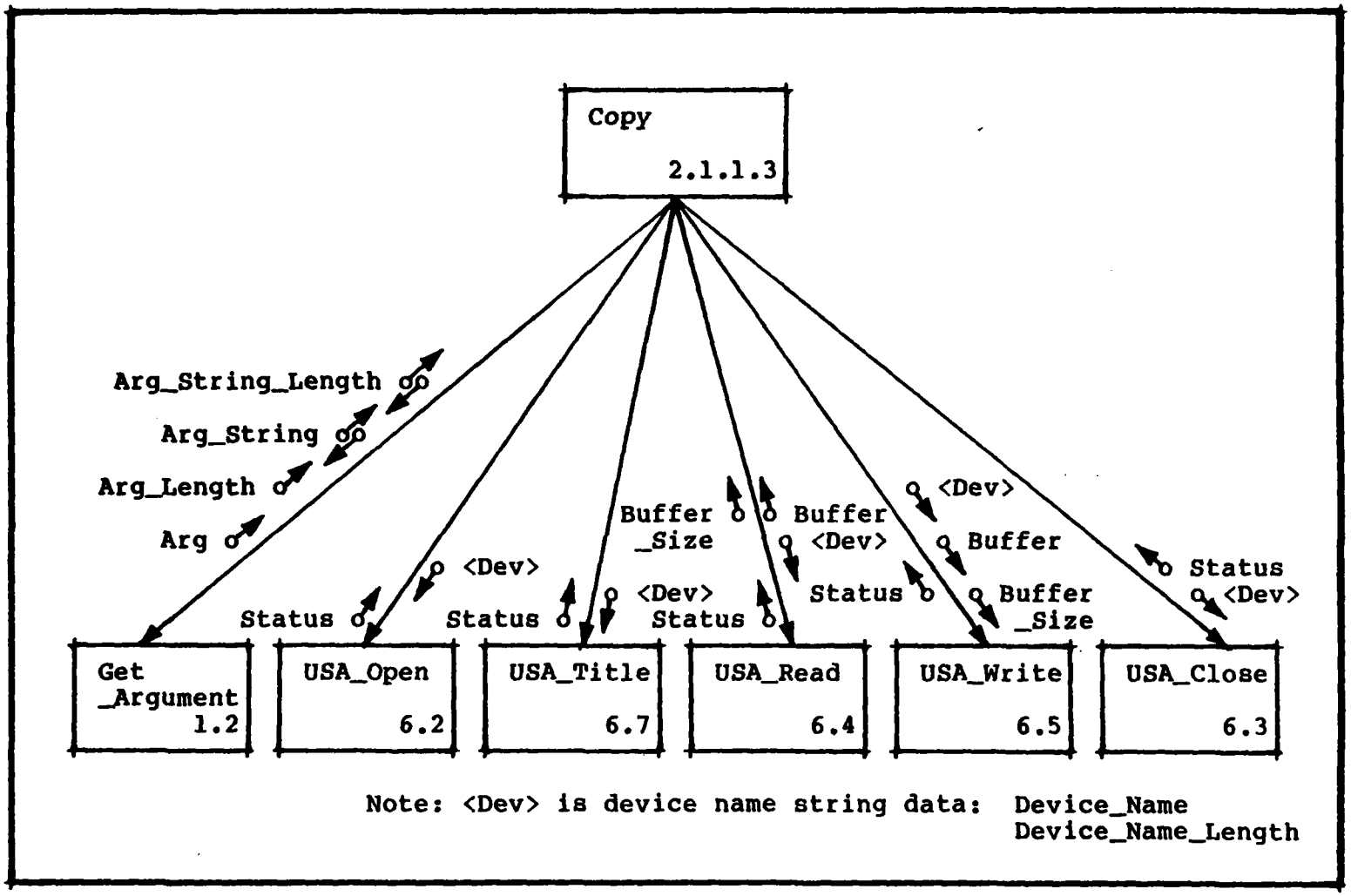


Figure J-8. Copy (System Commands) Structure Chart (2.1.1.3)

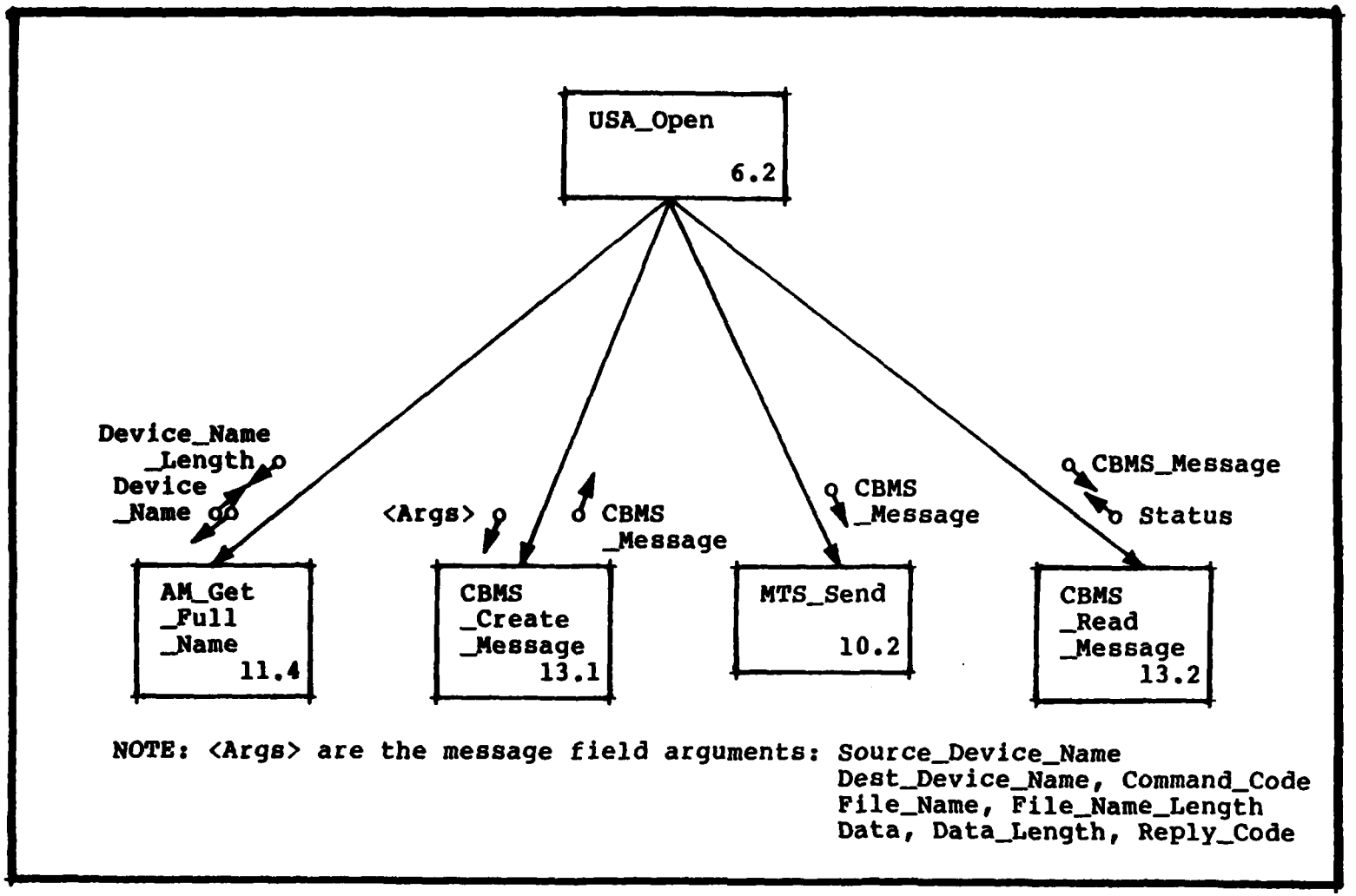


Figure J-9. USA_Open (User Shell Agent) Structure Chart (6.2)

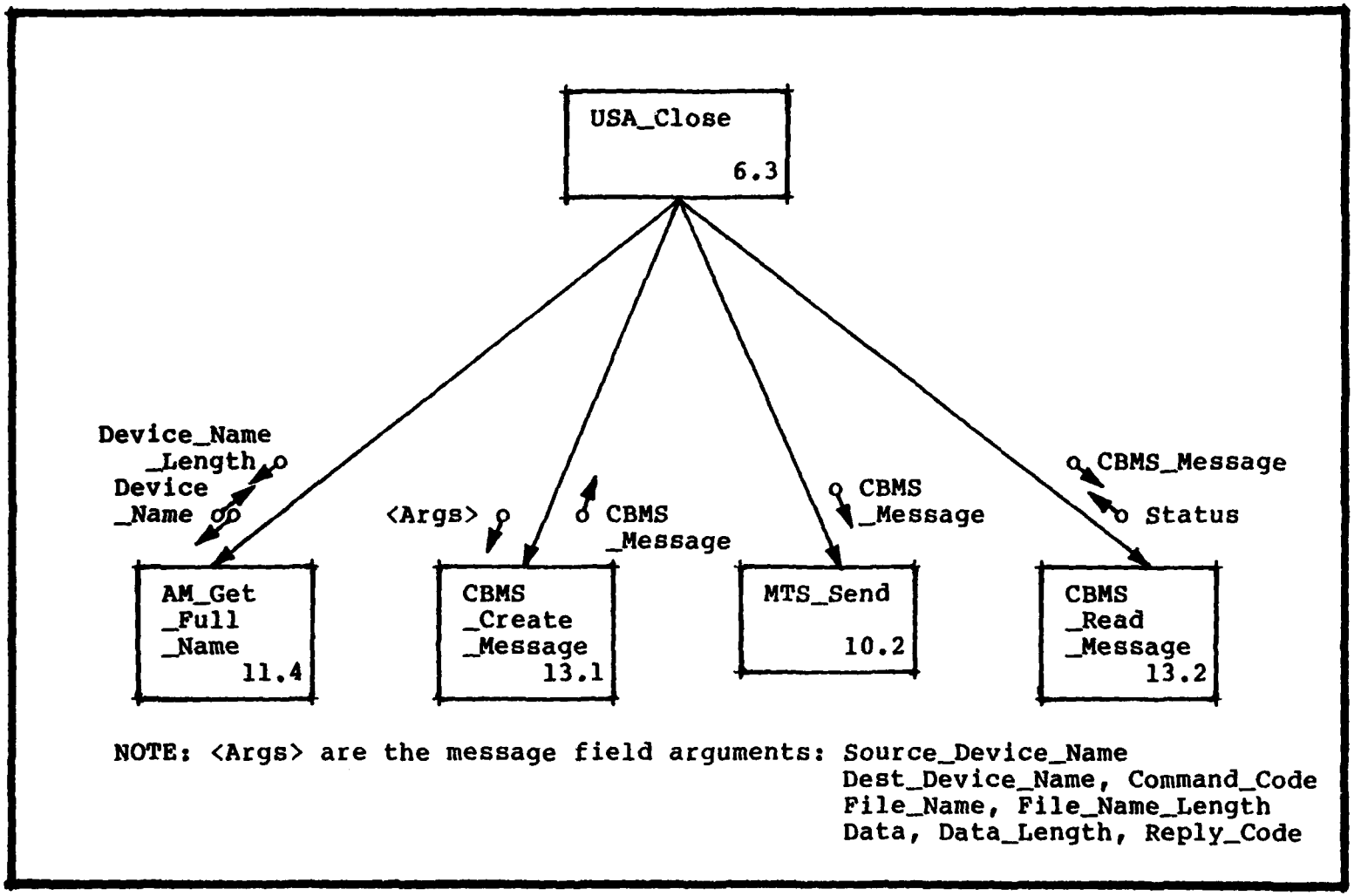


Figure J-10. USA_Close (User Shell Agent) Structure Chart (6.3)

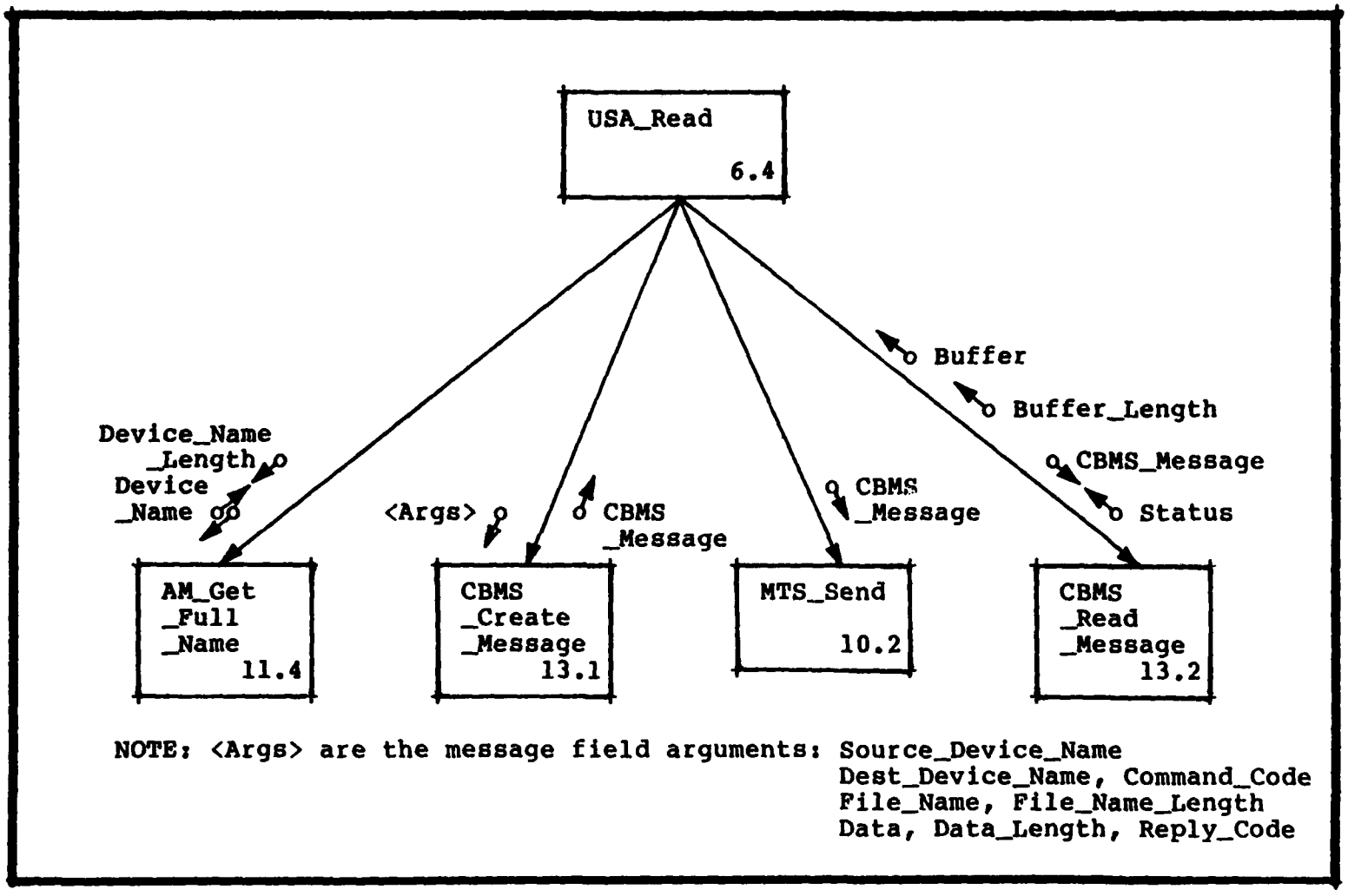


Figure J-11. USA_Read (User Shell Agent) Structure Chart (6.4)

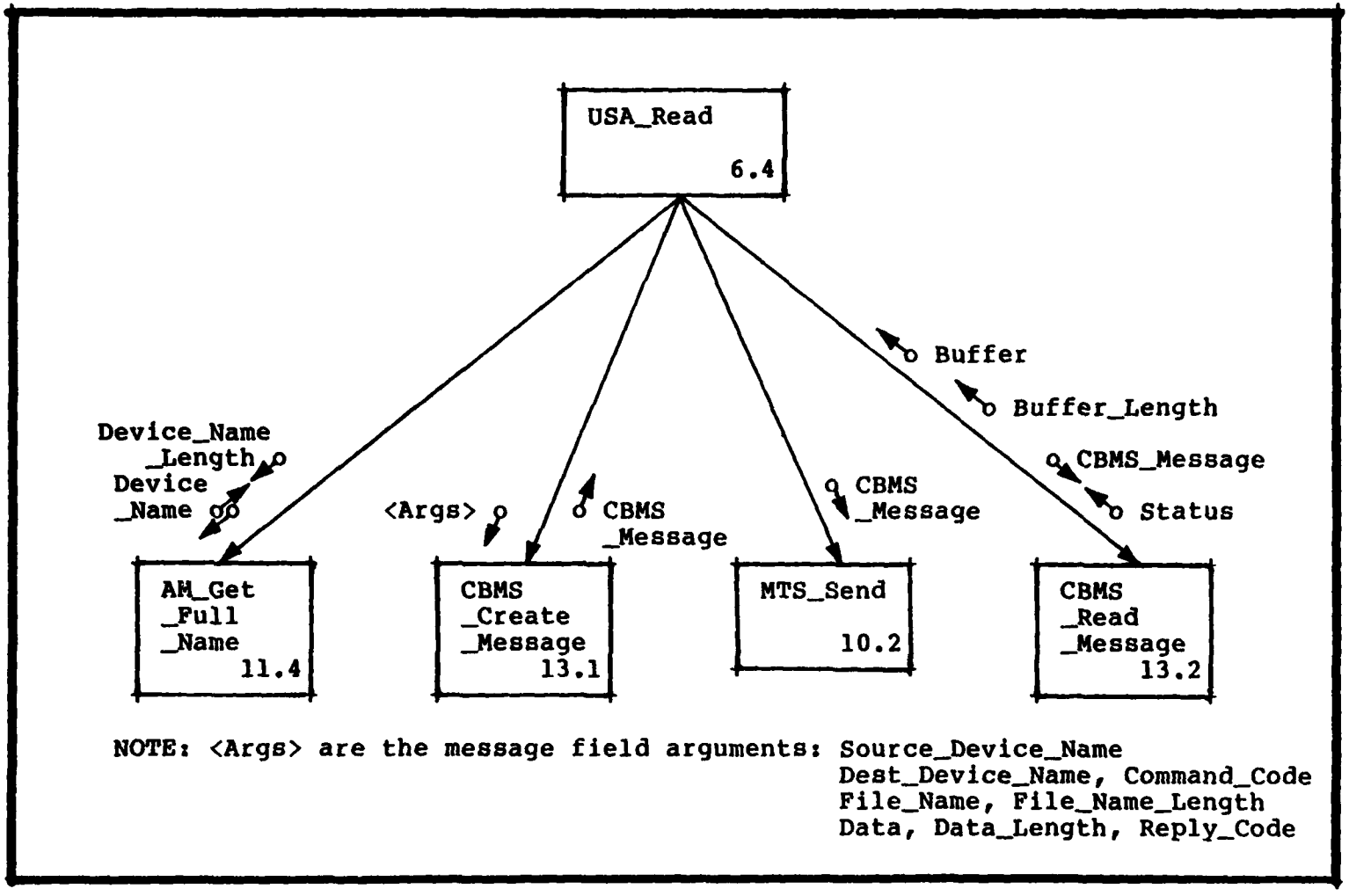


Figure J-11. USA_Read (User Shell Agent) Structure Chart (6.4)

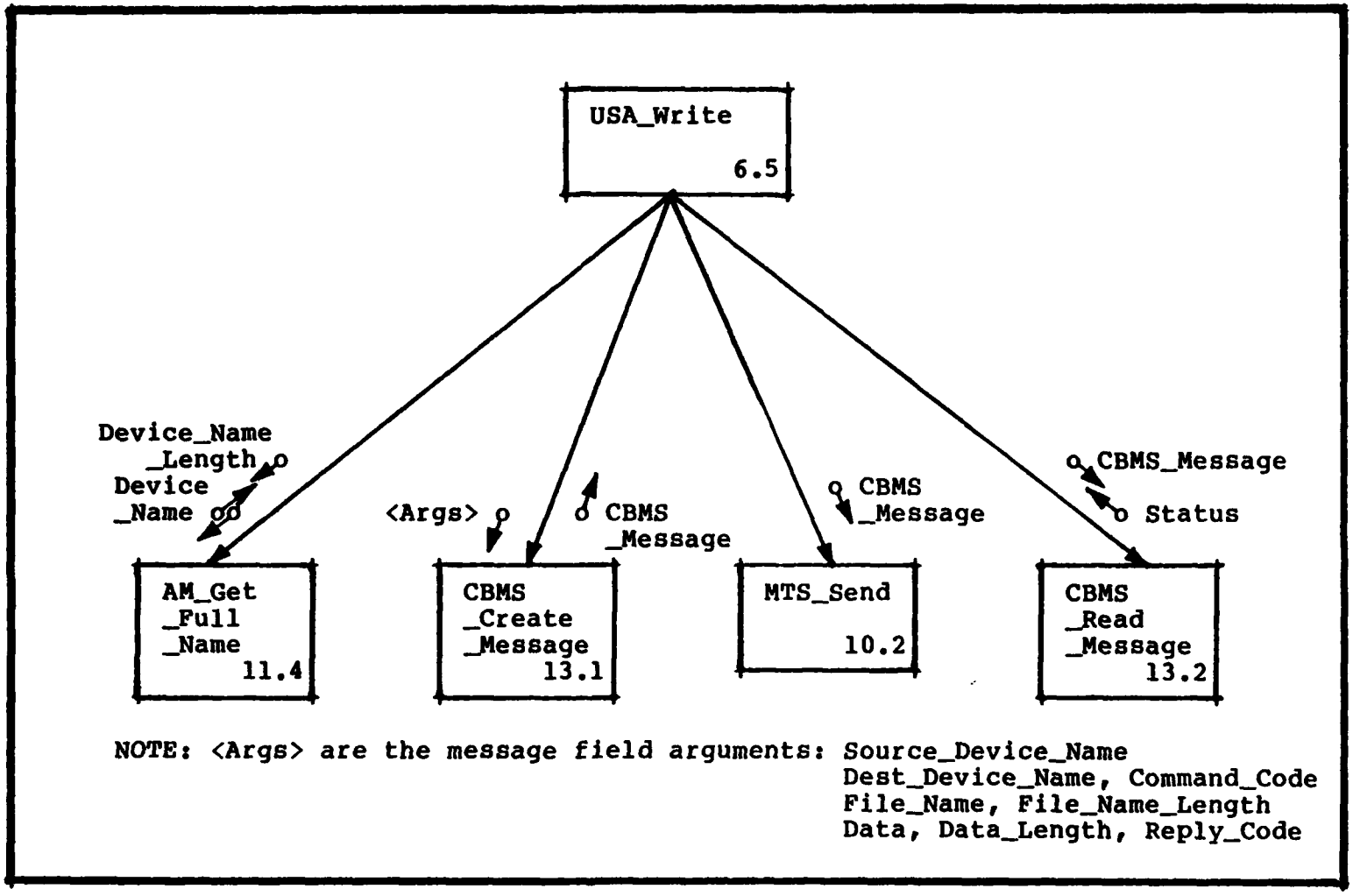


Figure J-12. USA_Write (User Shell Agent) Structure Chart (6.5)

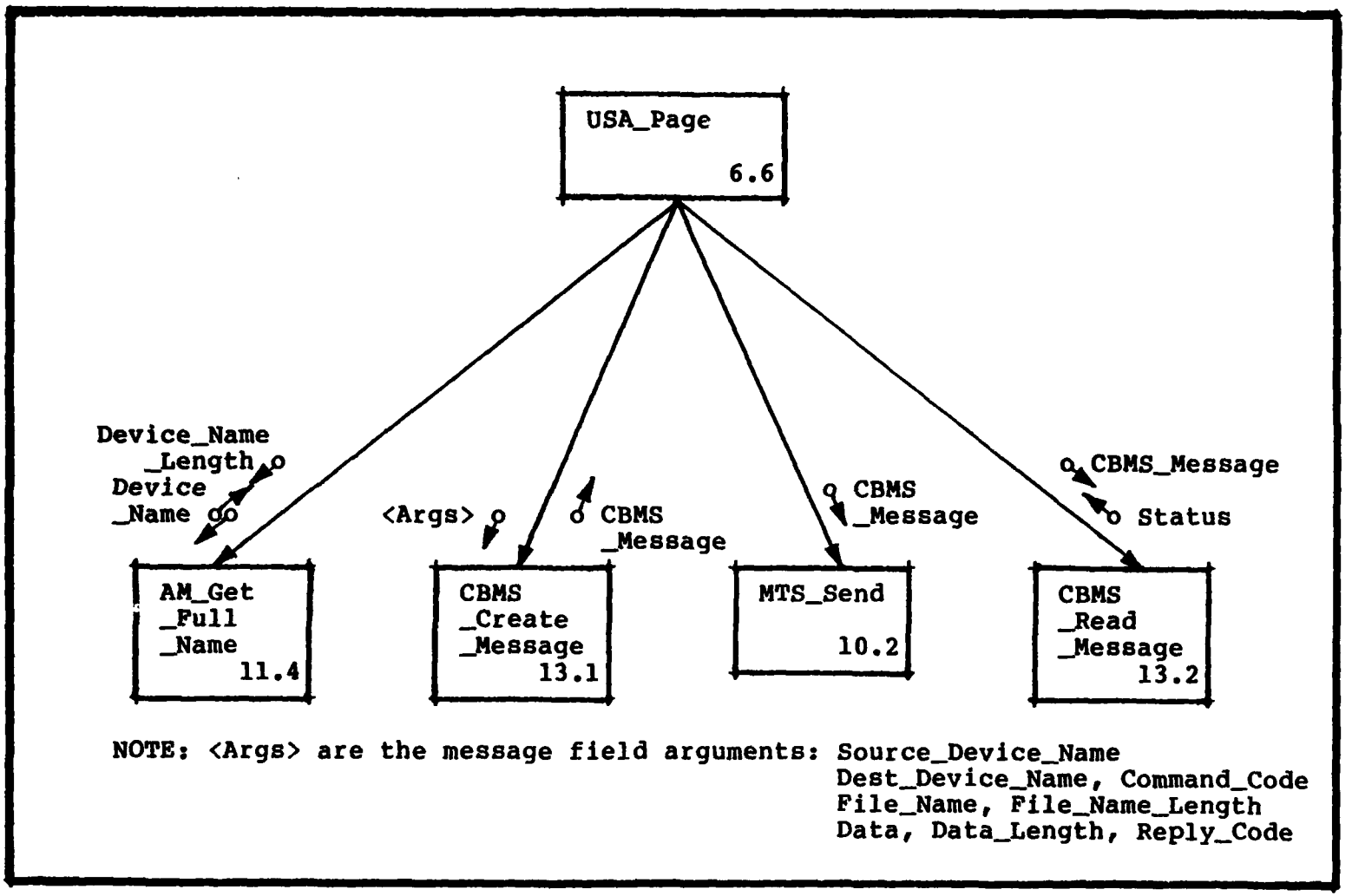


Figure J-13. USA_Page (User Shell Agent) Structure Chart (6.6)

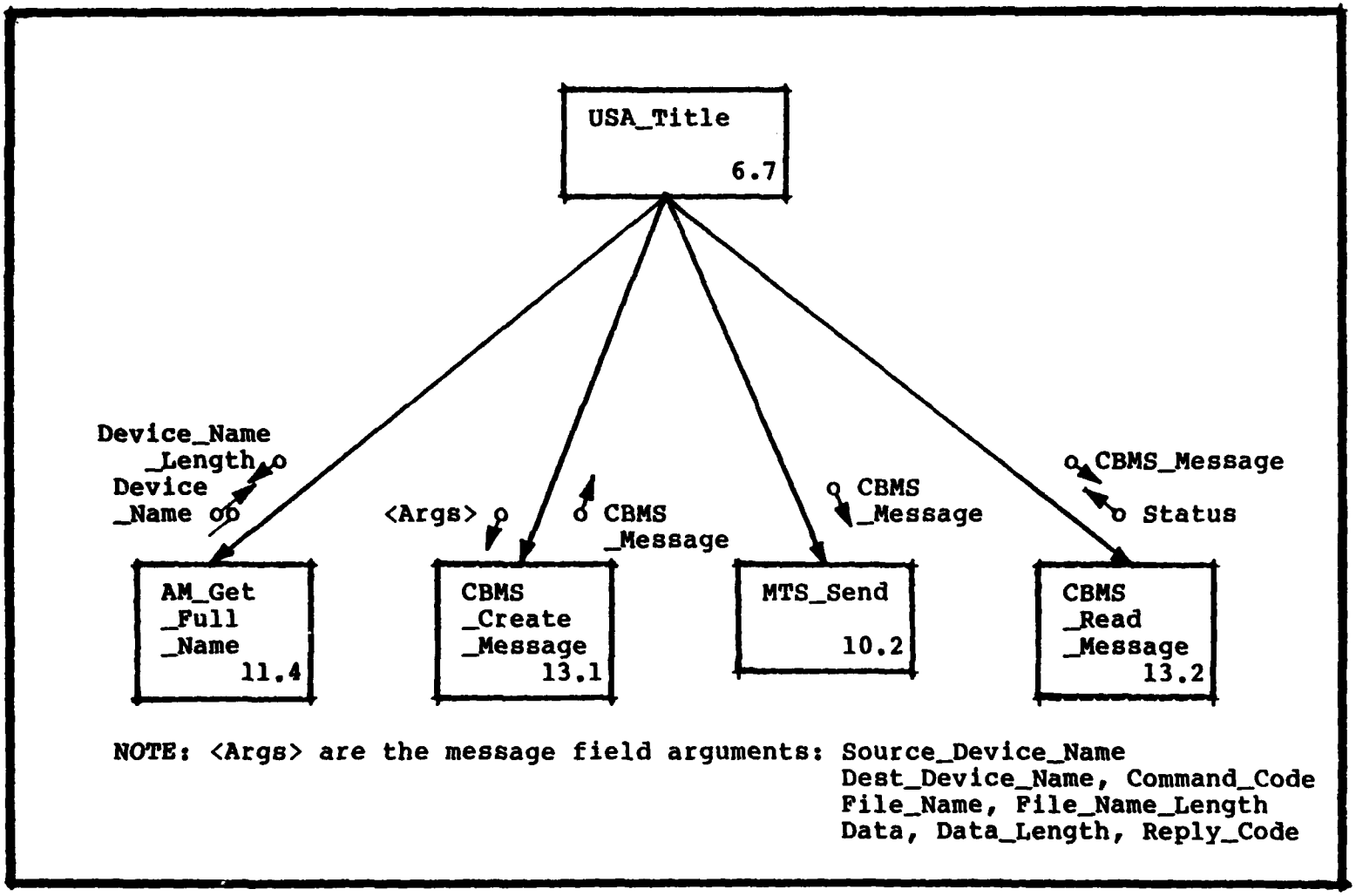


Figure J-14. USA_Title (User Shell Agent) Structure Chart (6.7)

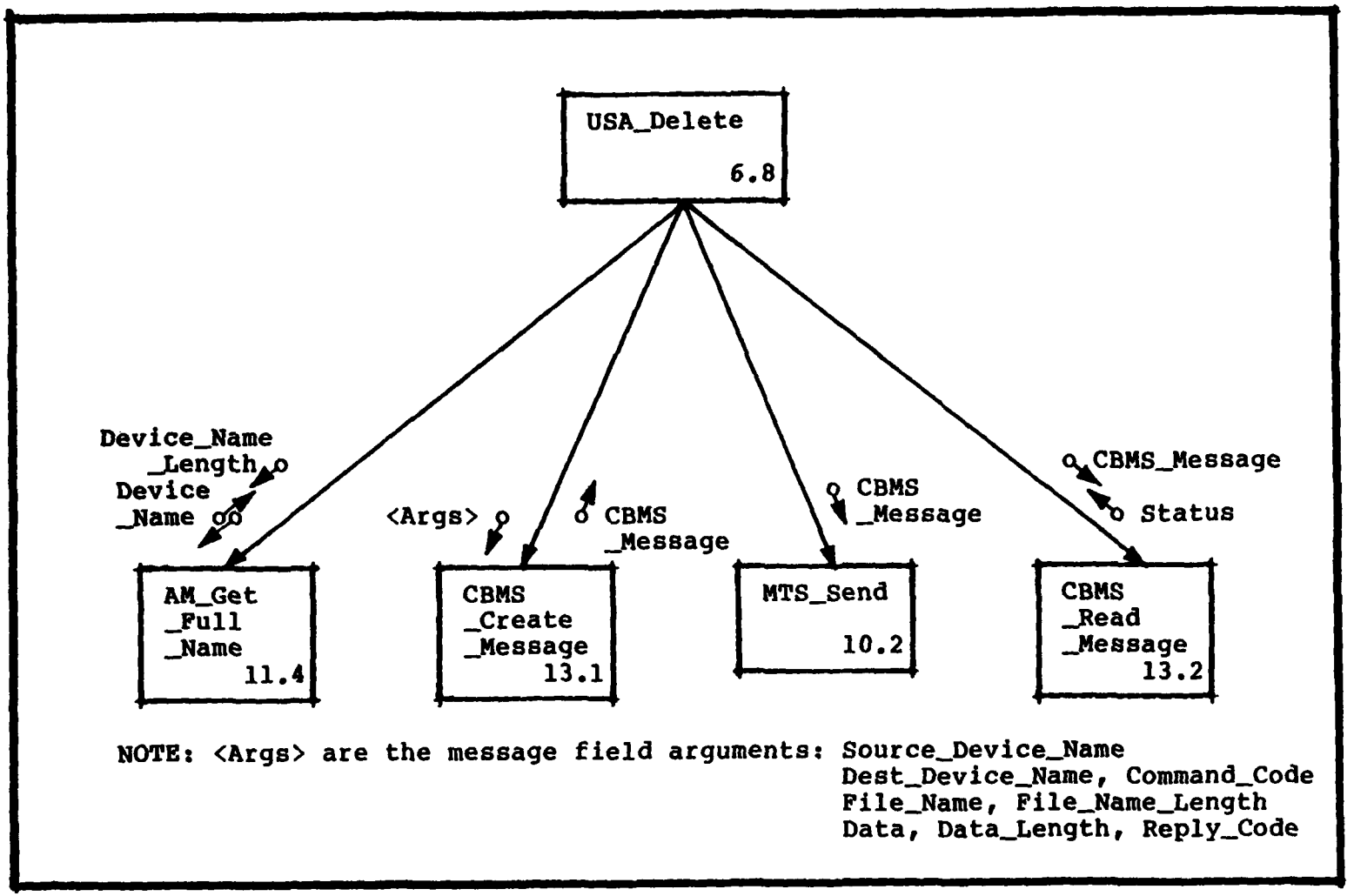


Figure J-15. USA_Delete (User Shell Agent) Structure Chart (6.8)

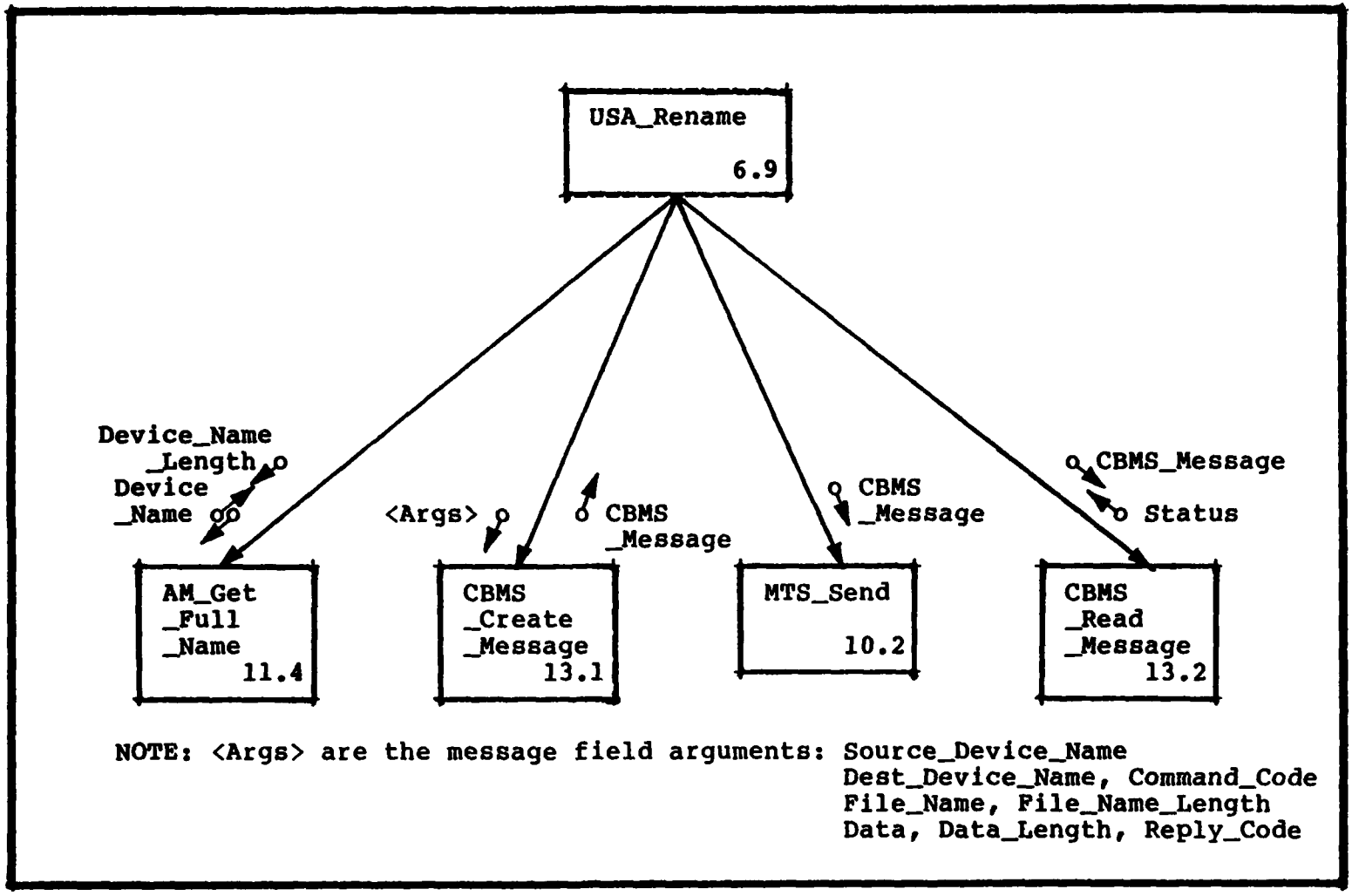


Figure J-16. USA_Rename (User Shell Agent) Structure Chart (6.9)

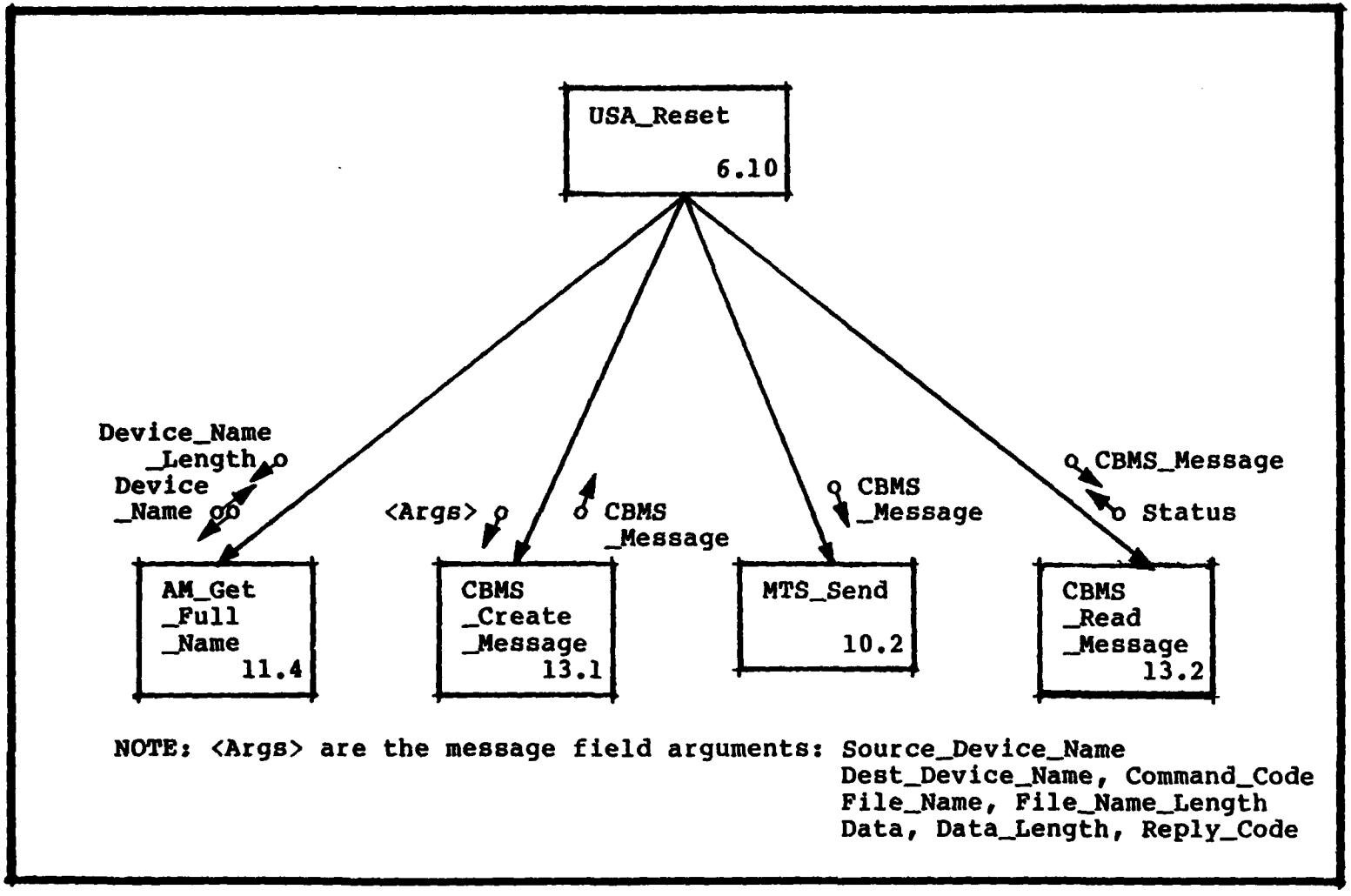


Figure J-17. USA_Reset (User Shell Agent) Structure Chart (6.10)

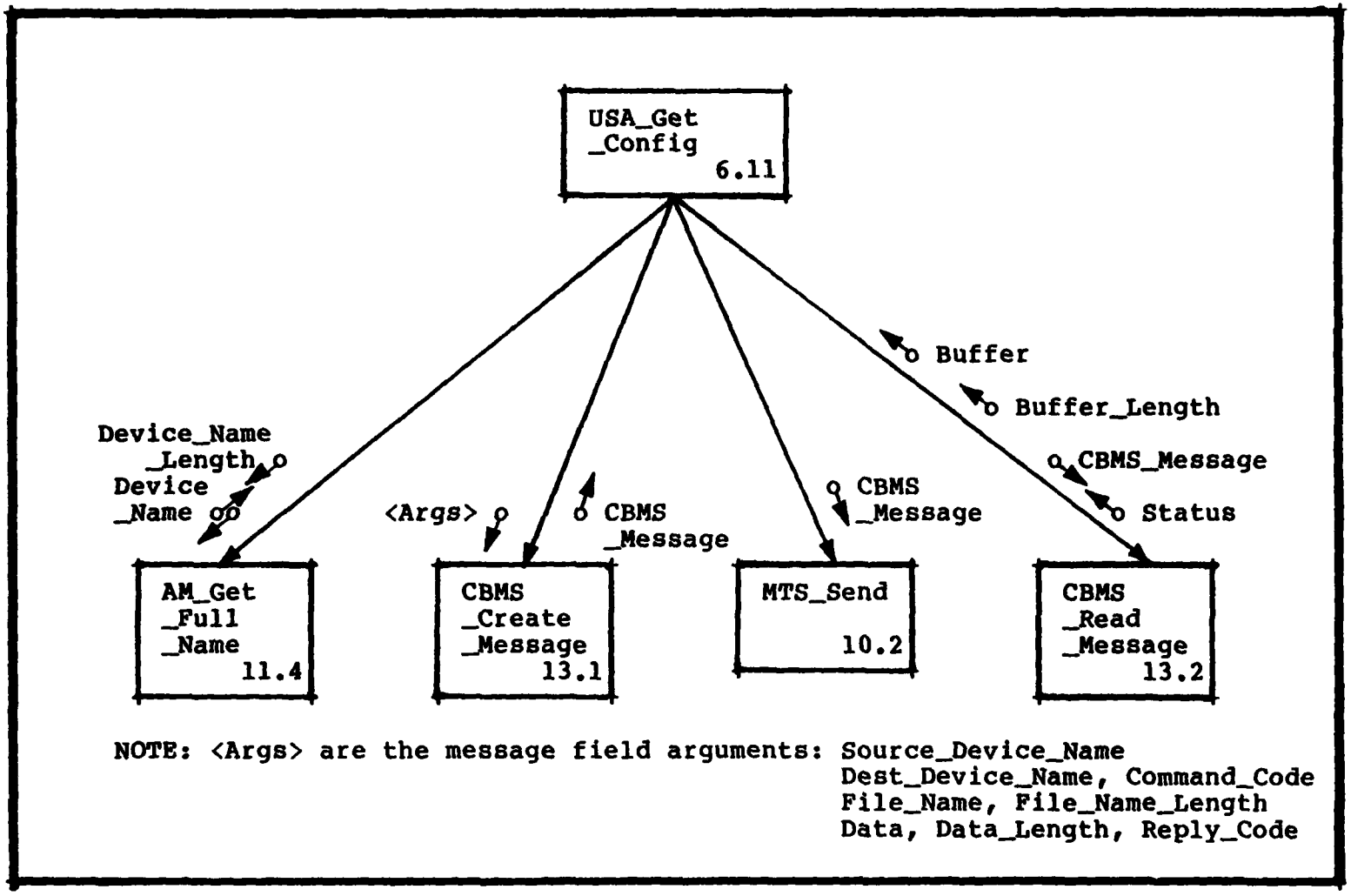


Figure J-18. USA_Get_Config (User Shell Agent) Structure Chart (6.11)

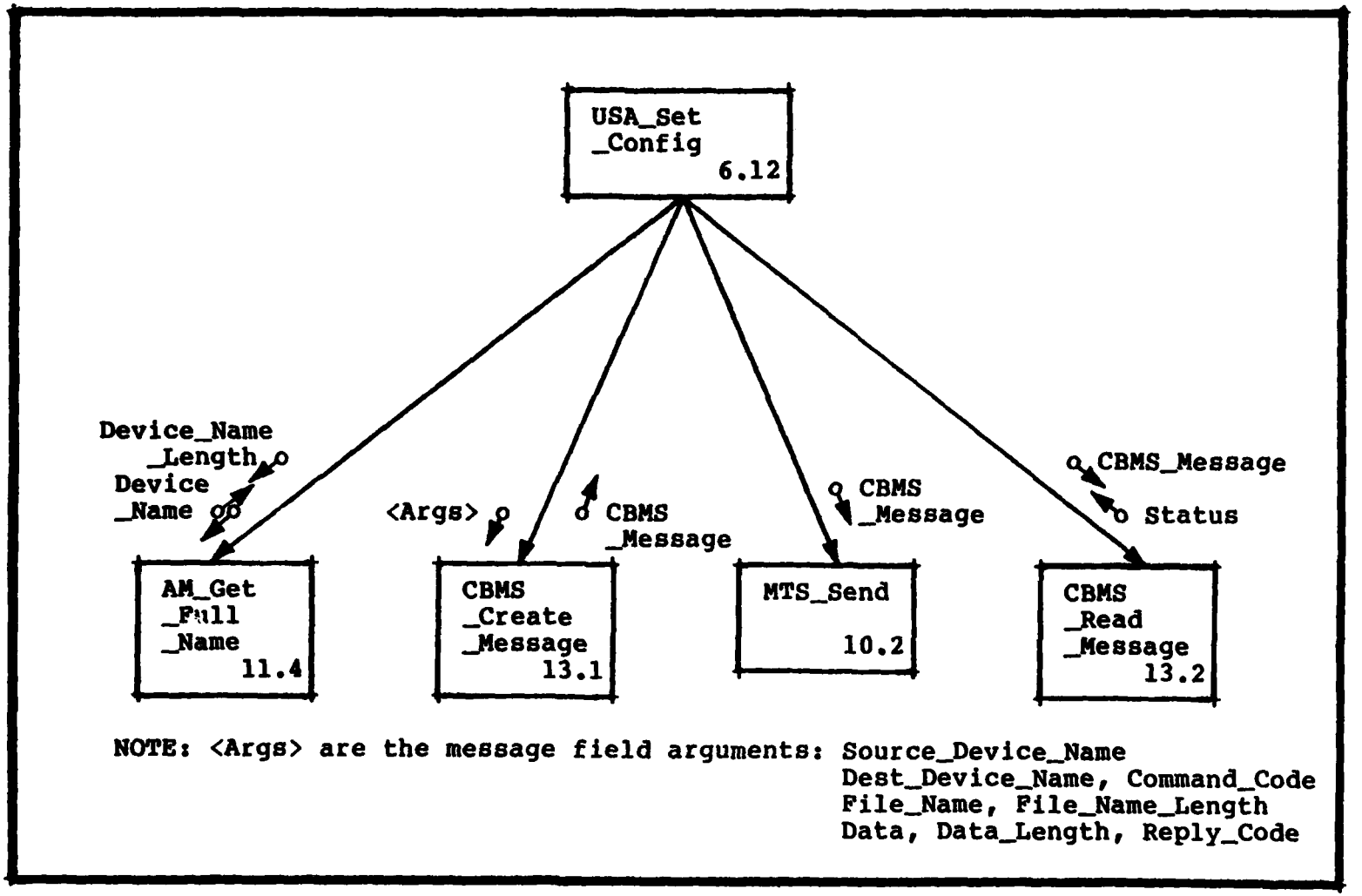


Figure J-19. USA_Set_Config (User Shell Agent) Structure Chart (6.12)

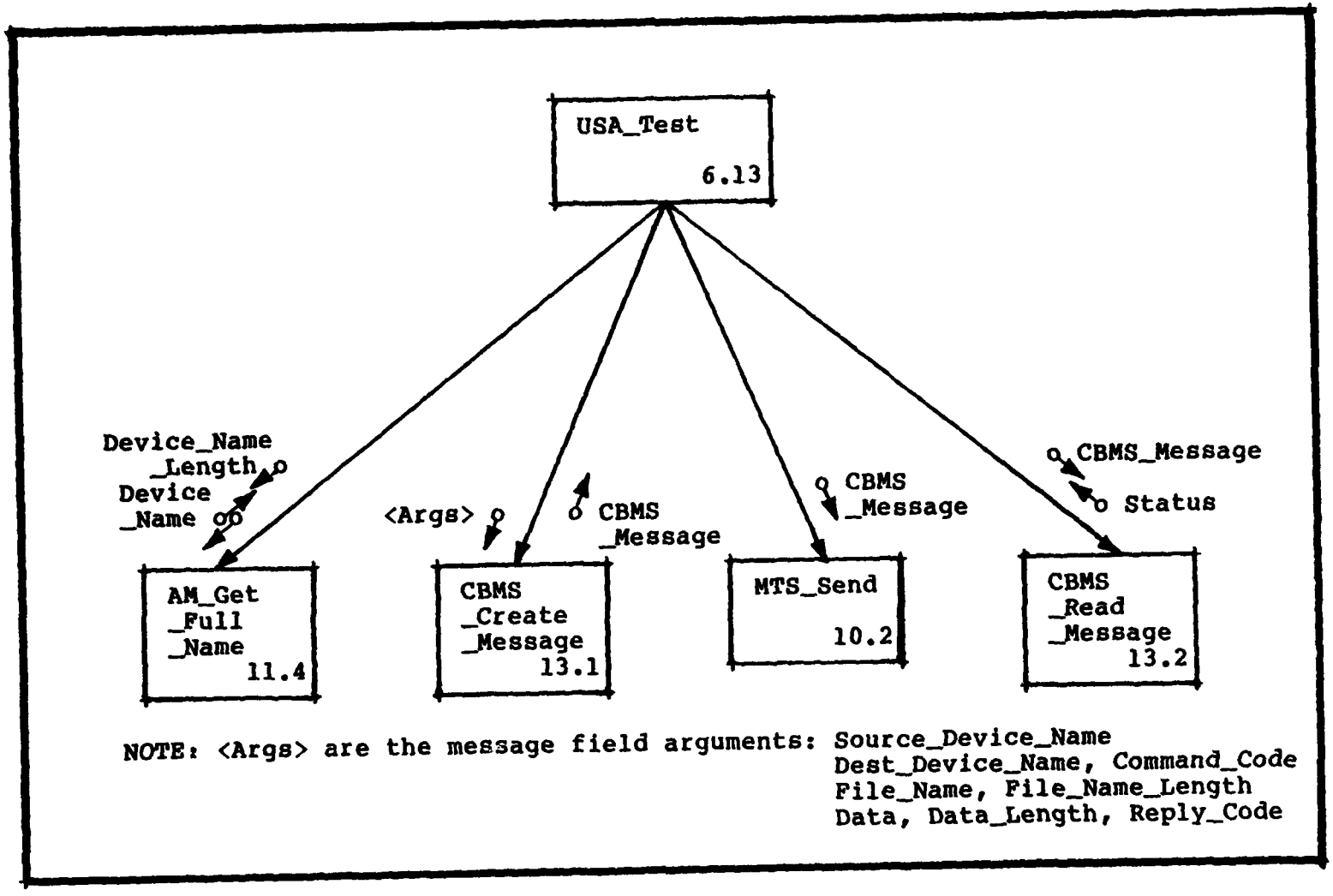


Figure J-20. USA_Test (User Shell Agent) Structure Chart (6.13)

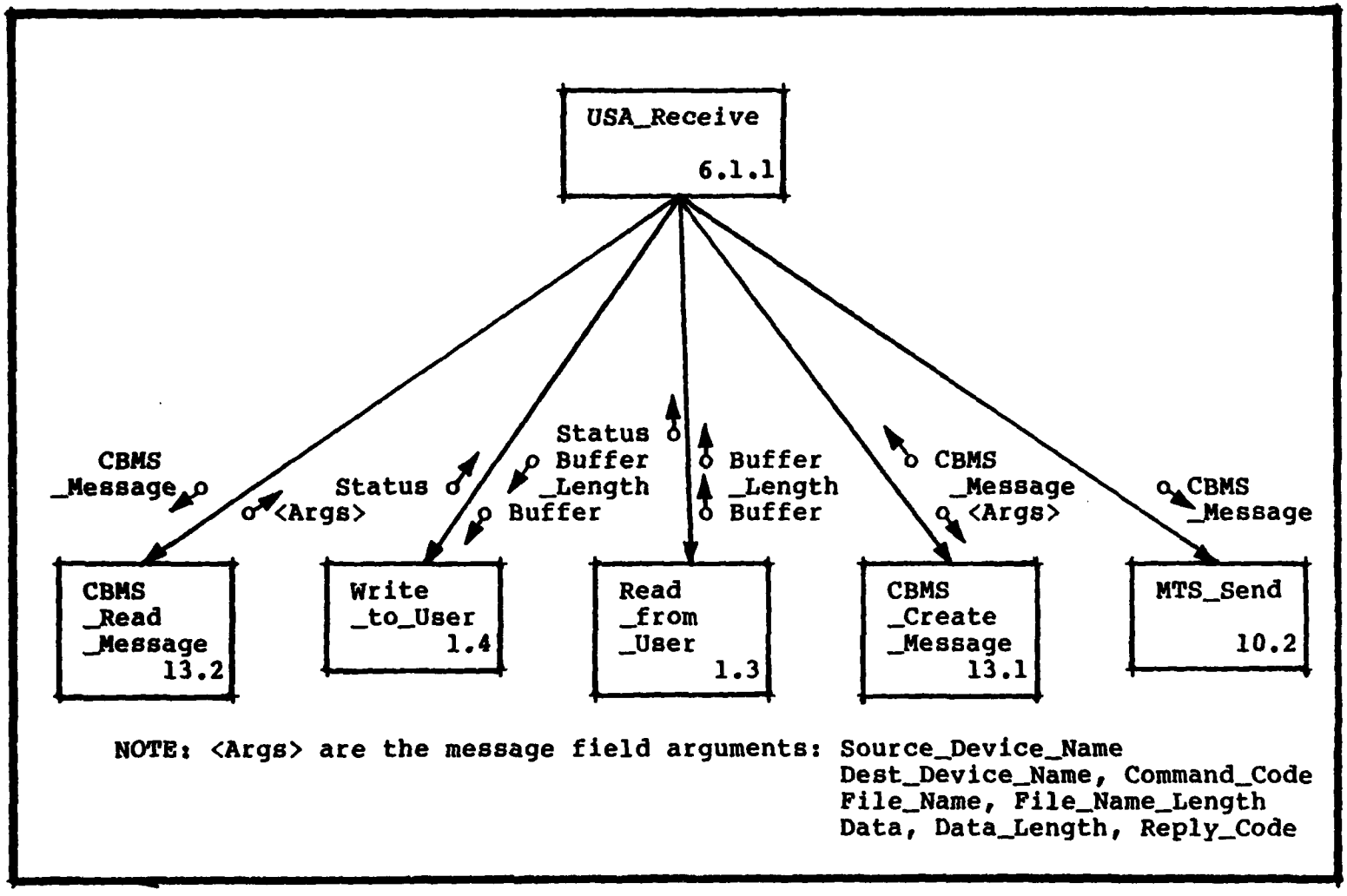


Figure J-21. USA_Receive (User Shell Agent) Structure Chart (6.1.1)

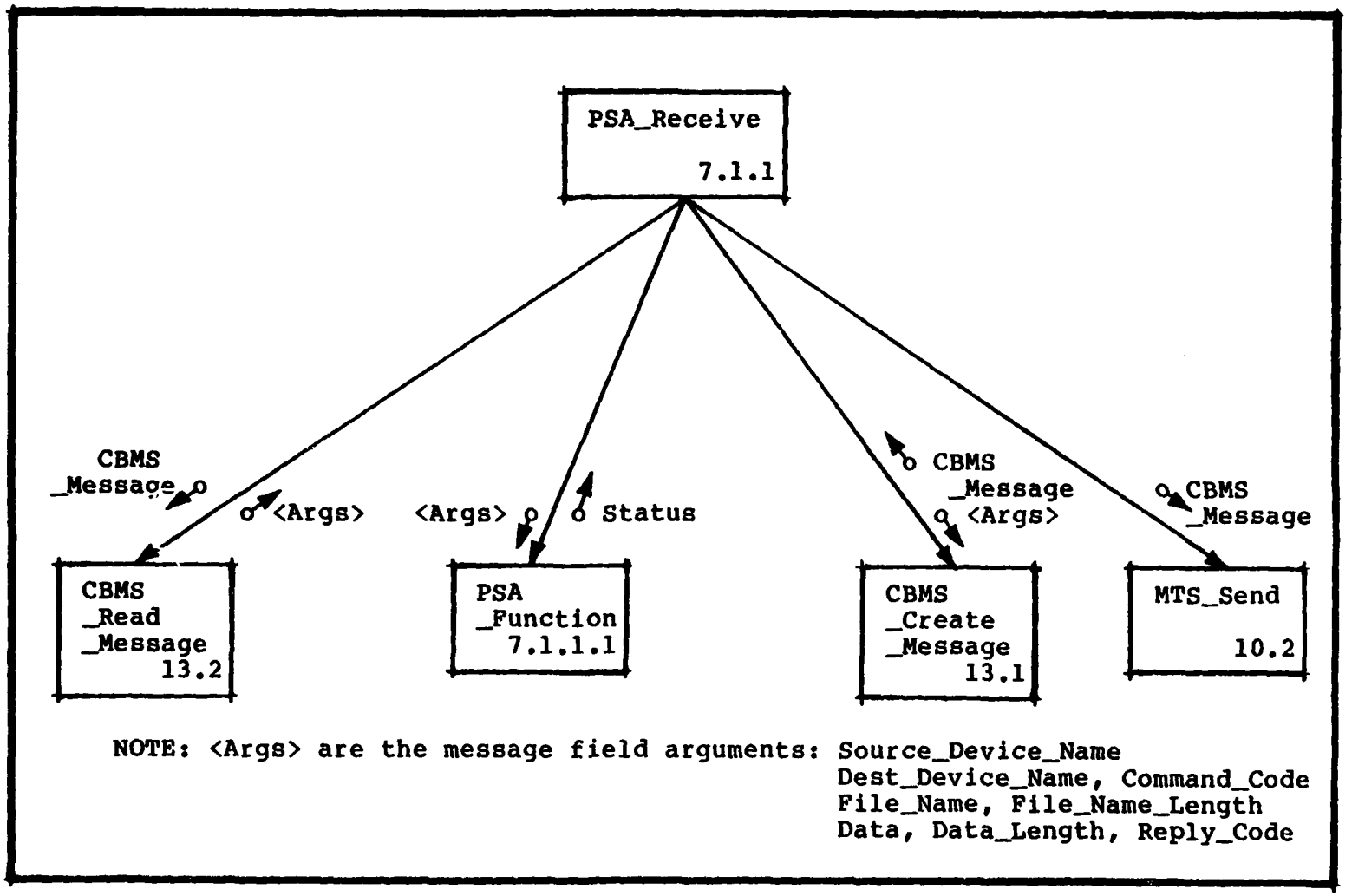


Figure J-22. PSA_Receive (Printer System Agent) Structure Chart (7.1.1)

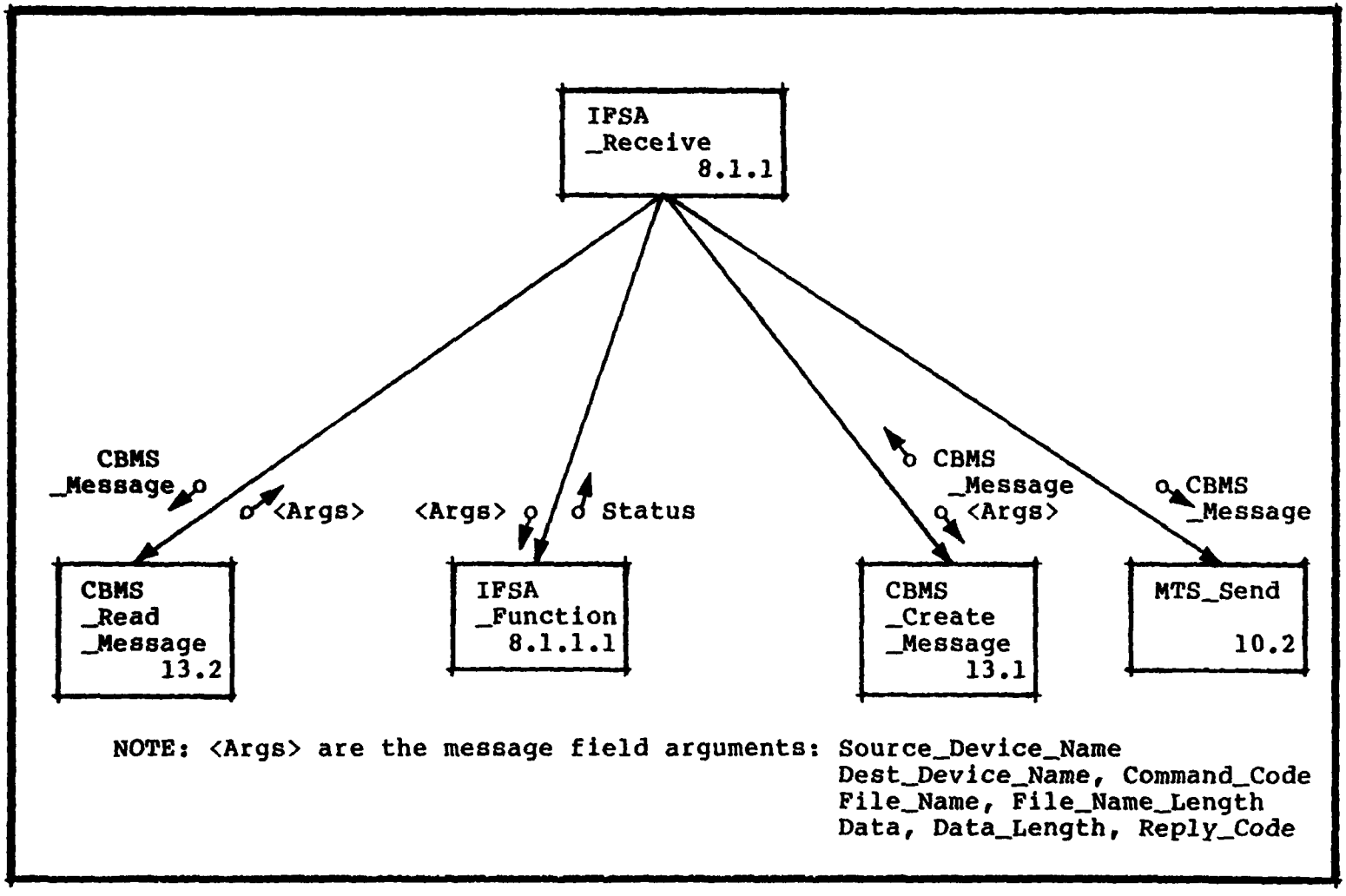
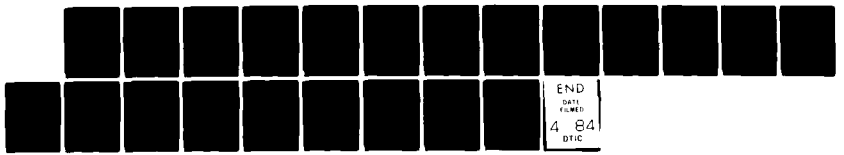


Figure J-23. IFSA_Receive (ISIS File System Agent) Structure Chart (8.1.1)

AD-A138 429 DESIGN AND IMPLEMENTATION OF AN INPUT/OUTPUT INTERFACE PROTOCOL FOR THE I..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. K N COLE

UNCLASSIFIED DEC 83 AFIT/GE/EE/83D-17 F/G 17/2 NL

4/4

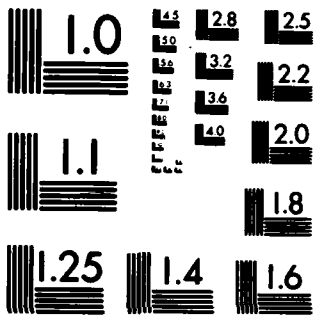


END

DATE FILMED

4 84

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

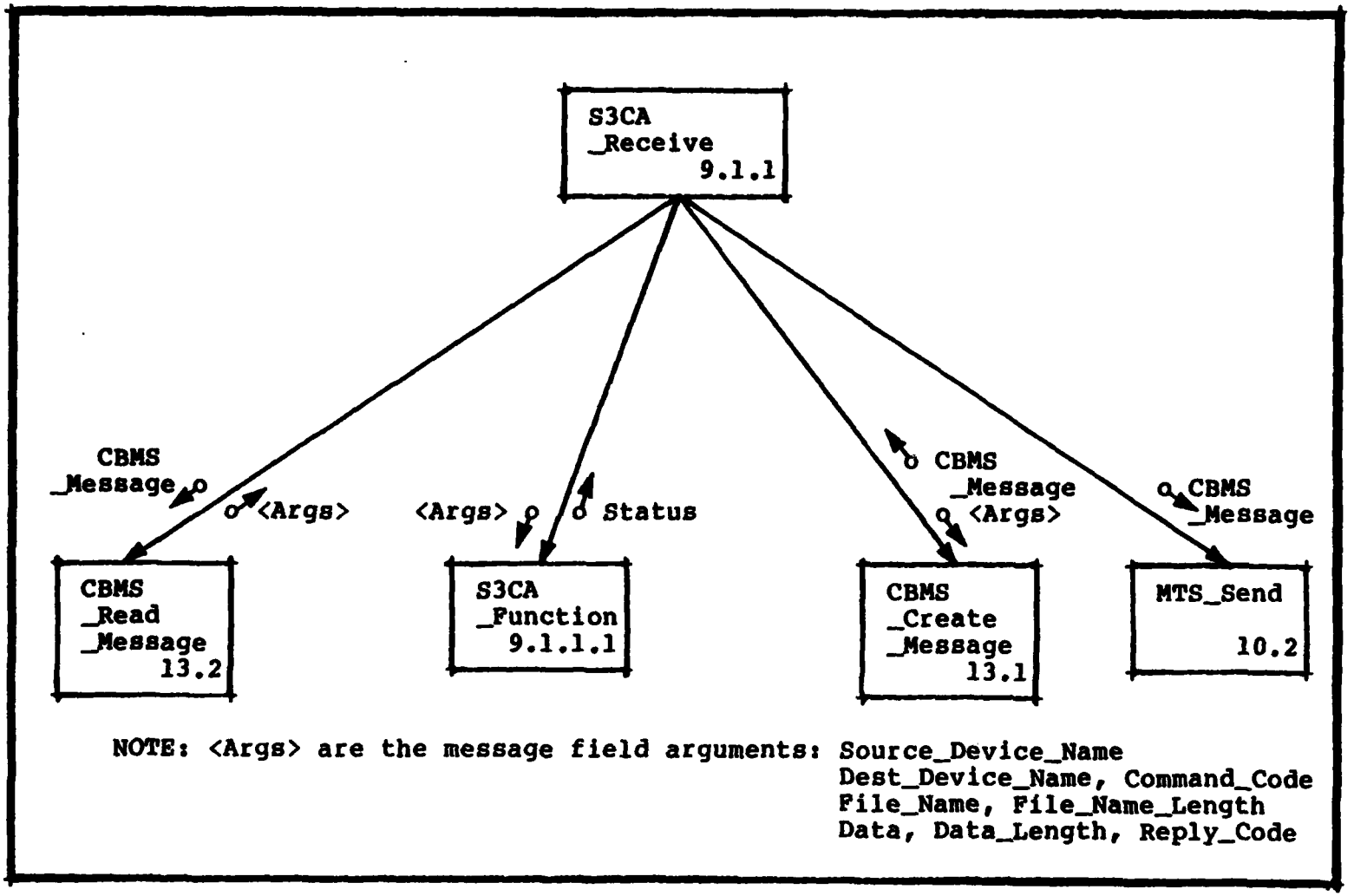


Figure J-24. S3CA_Receive (Series III Console Agent) Structure Chart (9.1.1)

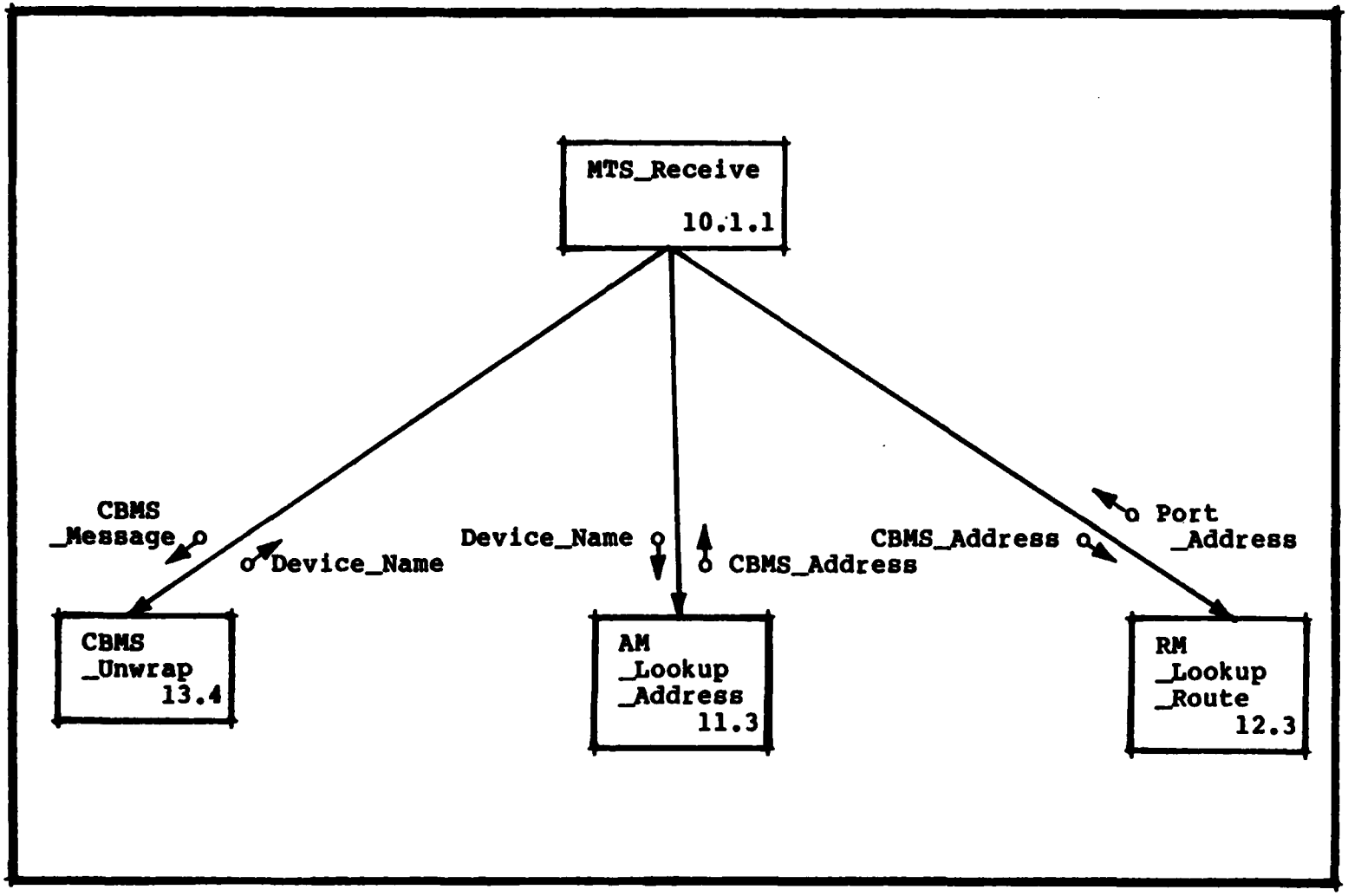


Figure J-25. MTS_Receive (Message Transfer System) Structure Chart (10.1.1)

APPENDIX K

Test Results

This appendix contains the results of the tests, described in Chapter IV, for the I/O Interface. These results are presented as a listing of the test command and the system response for each step of the procedure. The test procedures appear in the following sequence:

I. 432 Processor Validation

User Sublayer Validation Test (432)

User Agent Sublayer Validation Test (432)

Message Transfer Sublayer Validation Test (432)

II. Attached Processor Validation

User Sublayer Validation Test (AP)

User Agent Sublayer Validation Test (AP)

Message Transfer Sublayer Validation Test (AP)

III. System Integration Test

I. 432 Processor Validation

The description of the 432 Processor Validation Tests can be found in the body of this report (Chapter IV).

User Sublayer Validation Test (432)

The following is a listing of the information displayed on the Debugger console during the User Sublayer Validation Test. The system prompt (>) is followed by the operator's command entry. All command entries are under-lined.

```
>HELP
User Shell Commands:  HELP - Command Query
                      SET  - Default Naming
                      COPY - Data Transfer

>HELP SET
SET Command Use:  Default Device Naming
                  Argument may be appended to the left side
                    of abbreviated device name string

SET Command Syntax:  Command        Arguments
                       SET       DEFAULT   Device-Name
                        S

Valid Device Name:  <Country>/<Network>/<Host>/<Device>

     where  <Country> must be RM67
            <Network> must be NET0
            <Host> is one of [432, MDS]
            <Device> is one of [USR] on 432
                               [PTR, CON, DSK] on MDS

>HELP COPY
COPY Command Use:  Data Transfer from one system device
                     to another

COPY Command Syntax:  Command       Arguments
                        COPY    Device-Name-1 Device-Name-2
                         C

Valid Device Name:  <Country>/<Network>/<Host>/<Device>

     where  <Country> must be RM67
            <Network> must be NET0
            <Host> is one of [432, MDS]
            <Device> is one of [USR] on 432
                               [PTR, CON, DSK] on MDS
```

```
>SET DEFAULT RM67/NET0
RM_SET_DEFAULT: RM67/NET0

>COPY /432/USR /MDS/CON
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: OPEN
          DATA:
USA_SEND: DEVICE: RM67/NET0/MDS/CON
          FUNCTION: OPEN
          DATA:
USA_SEND: DEVICE: RM67/NET0/MDS/CON
          FUNCTION: TITLE
          DATA: RM67/NET0/432/USR
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: READ
          DATA:
USA_SEND: DEVICE: RM67/NET0/MDS/CON
          FUNCTION: WRITE
          DATA: THIS IS DUMMY TEST DATA
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: CLOSE
          DATA:
USA_SEND: DEVICE: RM67/NET0/MDS/CON
          FUNCTION: CLOSE
          DATA:

>COPY /MDS/CON /432/USR
USA_SEND: DEVICE: RM67/NET0/MDS/CON
          FUNCTION: OPEN
          DATA:
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: OPEN
          DATA:
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: TITLE
          DATA: RM67/NET0/432/USR
USA_SEND: DEVICE: RM67/NET0/MDS/CON
          FUNCTION: READ
          DATA:
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: WRITE
          DATA: THIS IS DUMMY TEST DATA
USA_SEND: DEVICE: RM67/NET0/MDS/CON
          FUNCTION: CLOSE
          DATA:
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: CLOSE
          DATA:
```

User Sublayer Validation Test (432) (continued)

```
>COPY /432/USR /432/USR
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: OPEN
          DATA:
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: OPEN
          DATA:
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: TITLE
          DATA: RM67/NET0/432/USR
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: READ
          DATA:
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: WRITE
          DATA: THIS IS DUMMY TEST DATA
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: CLOSE
          DATA:
USA_SEND: DEVICE: RM67/NET0/432/USR
          FUNCTION: CLOSE
          DATA:

>SET DEFAULT RM67/NET0/MDS
RM_SET_DEFAULT: RM67/NET0/MDS

>COPY /DSK/:F1:TEST.TXT /PTR
USA_SEND: DEVICE: RM67/NET0/MDS/DSK/:F1:TEST.TXT
          FUNCTION: OPEN
          DATA:
USA_SEND: DEVICE: RM67/NET0/MDS/PTR
          FUNCTION: OPEN
          DATA:
USA_SEND: DEVICE: RM67/NET0/MDS/PTR
          FUNCTION: TITLE
          DATA: RM67/NET0/MDS/DSK/:F1:TEST.TXT
USA_SEND: DEVICE: RM67/NET0/MDS/DSK/:F1:TEST.TXT
          FUNCTION: READ
          DATA:
USA_SEND: DEVICE: RM67/NET0/MDS/PTR
          FUNCTION: WRITE
          DATA: THIS IS DUMMY TEST DATA
USA_SEND: DEVICE: RM67/NET0/MDS/DSK/:F1:TEST.TXT
          FUNCTION: CLOSE
          DATA:
USA_SEND: DEVICE: RM67/NET0/MDS/PTR
          FUNCTION: CLOSE
          DATA:

>   (End of User Sublayer Validation Test (432))
```

User Agent Sublayer Validation Test (432)

This test was not performed due to the lack of sufficient disk storage on the Debugger system. The size of the executable code module exceeded the storage of a double density diskette.

Message Transfer Sublayer Validation Test (432)

This test was not performed due to the lack of sufficient disk storage on the Debugger system. The size of the executable code module exceeded the storage of a double density diskette.

II. Attached Processor Validation

The description of the Attached Processor Validation Tests can be found in the body of this report (Chapter IV).

User Sublayer Validation Test (AP)

The following is a listing of the test results of the Attached Processor system User Sublayer Validation Test. The operator inputs are under-lined. The name of the device, on which the output appears, is shown on the right (in brackets). Comments are in parentheses.

```
(Printer System Test) ------------------------------------

>0 (open)                                              [CON]
Device: PTR
Device Indicates: OK

>3 (write)
Device: PTR
Data: THIS IS A TEST OF THE PRINTER
THIS IS A TEST OF THE PRINTER                          [PTR]
Device Indicates: OK                                   [CON]

>4 (page)
Device: PTR
<form feed>                                            [PTR]
Device Indicates: OK                                   [CON]
```

```
>5 (title)
Device: PTR
Data: PRINTER TITLE PAGE TEST
<form feed>                                                    [PTR]
PRINTER TITLE PAGE TEST (centered on tenth line of the page)
<form feed>
Device Indicates: OK                                           [CON]

>9 (test)
Device: PTR
Device Indicates: OK

>1 (close)
Device: PTR
Device Indicates: OK


(ISIS File System Test) ----------------------------------

>0 (open)                                                      [CON]
Device: DSK/:F1:TEST.TXT
Device Indicates: OK

>3 (write)
Device: DSK/:F1:TEST.TXT
Data: THIS IS A TEST OF THE ISIS FILE SYSTEM
THIS IS A TEST OF THE ISIS FILE SYSTEM          [DSK/:F1:TEST.TXT]
                          (verified by using ISIS utilities)
Device Indicates: OK                                           [CON]

>1 (close)
Device: DSK/:F1:TEST.TXT
Device Indicates: OK

>0 (open)
Device: DSK/:F1:TEST.TXT
Device Indicates: OK

>2 (read)
Device: DSK/:F1:TEST.TXT
Data:  P ("<ctrl>-@ P" was the two character sequence
                  entered to indicated a buffer length of 80)
THIS IS A TEST OF THE ISIS FILE SYSTEM
Device Indicates: END-OF-FILE

>2 (read)
Device: DSK/:F1:TEST.TXT
Data:  P ("<ctrl>-@ P" was the two character sequence
                  entered to indicated a buffer length of 80)
Device Indicates: END-OF-FILE
```

>1 (close)
Device: DSK/:F1:TEST.TXT
Device Indicates: OK

>7 (rename)
Device: DSK/:F1:TEST.TXT
Data: :F1:NEW.TXT
Device Indicates: OK

>6 (delete)
Device: DSK/:F1:NEW.TXT
Device Indicates: OK

>8 (reset)
Device: DSK
Device Indicates: OK

>9 (test)
Device: DSK
Device Indicates: OK

(Series III Console Test) ----------

>2 (read) [CON]
Device: CON
Data: P ("<ctrl>-@ P" was the two character sequence
entered to indicated a buffer length of 80)
THIS IS A TEST OF READING THE CONSOLE
THIS IS A TEST OF READING THE CONSOLE
Device Indicates: OK

>3 (write)
Device: CON
Data: THIS IS A TEST OF WRITING TO THE CONSOLE
THIS IS A TEST OF WRITING TO THE CONSOLE
Device Indicates: OK

>4 (page)
Device: CON
(the screen cleared and the cursor appeared in the upper left)
Device Indicates: OK

>5 (title)
Device: CON
(the screen cleared and the cursor appeared in the upper left)
Device Indicates: OK

User Sublayer Validation Test - Series III Console (continued)

```
>9 (test)
Device: CON
Device Indicates: OK

(End of User Sublayer Validation Test)
```

User Agent Sublayer Validation Test (AP)

The following is a listing of the test results of the Attached Processor system User Agent Sublayer Validation Test. The operator inputs are under-lined. The name of the device, on which the output appears, is shown on the right (in brackets). Comments are in parentheses.

```
(Printer System Test) ----------------------------------

>0 (open)                                                [CON]
Device: PTR
Device Indicates: OK

>2 (read)
Device: PTR
Data:  P ("<ctrl>-@ P" was the two character sequence
                entered to indicated a buffer length of 80)
Device Indicates: Invalid Command

>3 (write)
Device: PTR
Data: THIS IS A TEST OF THE PRINTER
THIS IS A TEST OF THE PRINTER                            [PTR]
Device Indicates: OK                                     [CON]

>4 (page)
Device: PTR
<form feed>                                              [PTR]
Device Indicates: OK                                     [CON]

>5 (title)
Device: PTR
Data: PRINTER TITLE PAGE TEST
<form feed>                                              [PTR]
PRINTER TITLE PAGE TEST (centered on tenth line of the page)
<form feed>
Device Indicates: OK                                     [CON]

>1 (close)
Device: PTR
Device Indicates: OK
```

```
>6 (delete)
Device: PTR
Device Indicates: Invalid Command

>7 (rename)
Device: PTR
Data: PPP
Device Indicates: Invalid Command

>9 (test)
Device: PTR
Device Indicates: OK

>G (get config)
Device: PTR
Device Indicates: Invalid Command

>S (set config)
Device: PTR
Data: 00000000
Device Indicates: OK


(ISIS File System Test) ----------------------------------

>0 (open)                                              [CON]
Device: DSK/:F1:TEST.TXT
Device Indicates: OK

>4 (page)
Device: DSK/:F1:TEST.TXT
Device Indicates: OK

>5 (title)
Device: DSK/:F1:TEST.TXT
Data: TEST OF TITLE ON ISIS FILE SYSTEM
Device Indicates: OK

>3 (write)
Device: DSK/:F1:TEST.TXT
Data: THIS IS A TEST OF THE ISIS FILE SYSTEM
THIS IS A TEST OF THE ISIS FILE SYSTEM       [DSK/:F1:TEST.TXT]
                          (verified by using ISIS utilities)
Device Indicates: OK                                   [CON]

>1 (close)
Device: DSK/:F1:TEST.TXT
Device Indicates: OK
```

```
>0 (open)
Device: DSK/:F1:TEST.TXT
Device Indicates: OK

>2 (read)
Device: DSK/:F1:TEST.TXT
Data:  P ("<ctrl>-@ P" was the two character sequence
                entered to indicated a buffer length of 80)
THIS IS A TEST OF THE ISIS FILE SYSTEM
Device Indicates: END-OF-FILE

>2 (read)
Device: DSK/:F1:TEST.TXT
Data:  P ("<ctrl>-@ P" was the two character sequence
                entered to indicated a buffer length of 80)
Device Indicates: END-OF-FILE

>1 (close)
Device: DSK/:F1:TEST.TXT
Device Indicates: OK

>7 (rename)
Device: DSK/:F1:TEST.TXT
Data: :F1:NEW.TXT
Device Indicates: OK

>6 (delete)
Device: DSK/:F1:NEW.TXT
Device Indicates: OK

>8 (reset)
Device: DSK
Device Indicates: OK

>9 (test)
Device: DSK
Device Indicates: OK

>G (get config)
Device: DSK
Device Indicates: Invalid Command

>S (set config)
Device: DSK
Data: 00000000
Device Indicates: Invalid Command
```

User Agent Sublayer Validation Test (continued)

(Series III Console Test) ----------

>0 (open) [CON]
Device: CON
Device Indicates: OK

>1 (close)
Device: CON
Device Indicates: OK

>2 (read)
Device: CON
Data: P ("<ctrl>-@ P" was the two character sequence
entered to indicated a buffer length of 80)
THIS IS A TEST OF READING THE CONSOLE
THIS IS A TEST OF READING THE CONSOLE
Device Indicates: OK

>3 (write)
Device: CON
Data: THIS IS A TEST OF WRITING TO THE CONSOLE
THIS IS A TEST OF WRITING TO THE CONSOLE
Device Indicates: OK

>4 (page)
Device: CON
(the screen cleared and the cursor appeared in the upper left)
Device Indicates: OK

>5 (title)
Device: CON
(the screen cleared and the cursor appeared in the upper left)
Device Indicates: OK

>6 (delete)
Device: CON
Device Indicates: Invalid Command

>7 (rename)
Device: CON
Data: PPP
Device Indicates: Invalid Command

>8 (reset)
Device: CON
Device Indicates: OK

>9 (test)
Device: CON
Device Indicates: OK

```
>G (get config)
Device: CON
Device Indicates: Invalid Command

>S (set config)
Device: CON
Data: 00000000
Device Indicates: OK

(End of User Agent Sublayer Validation Test)
-------------------------------------------------------------
```

Message Transfer Sublayer Validation Test (AP)

The following is a listing of the test results of the Attached Processor system Message Transfer Sublayer Validation Test. The operator inputs are under-lined. The name of the device, on which the output appears, is shown on the right (in brackets). Comments are in parentheses.

Note that these results are exactly the same as for the User Agent Sublayer Validation test (above). There is no apparent change in the operation of the system. The difference is in the fact that the commands are now sent through the Message Transfer Sublayer routing mechanism to reach the appropriate device. In the previous test, the test shell gave the command message directly to the User Agent for the device requested.

```
(Printer System Test) ------------------------------------

>0 (open)                                              [CON]
Device: PTR
Device Indicates: OK

>2 (read)
Device: PTR
Data:  P ("<ctrl>-@ P" was the two character sequence
                entered to indicated a buffer length of 80)
Device Indicates: Invalid Command

>3 (write)
Device: PTR
Data: THIS IS A TEST OF THE PRINTER
THIS IS A TEST OF THE PRINTER                          [PTR]
Device Indicates: OK                                   [CON]
```

```
>4 (page)
Device: PTR
<form feed>                                              [PTR]
Device Indicates: OK                                     [CON]

>5 (title)
Device: PTR
Data: PRINTER TITLE PAGE TEST
<form feed>                                              [PTR]
PRINTER TITLE PAGE TEST (centered on tenth line of the page)
<form feed>
Device Indicates: OK                                     [CON]

>1 (close)
Device: PTR
Device Indicates: OK

>6 (delete)
Device: PTR
Device Indicates: Invalid Command

>7 (rename)
Device: PTR
Data: PPP
Device Indicates: Invalid Command

>2 (test)
Device: PTR
Device Indicates: OK

>G (get config)
Device: PTR
Device Indicates: Invalid Command

>S (set config)
Device: PTR
Data: 00000000
Device Indicates: OK


(ISIS File System Test) ----------------------------------

>0 (open)                                                [CON]
Device: DSK/:F1:TEST.TXT
Device Indicates: OK

>4 (page)
Device: DSK/:F1:TEST.TXT
Device Indicates: OK
```

Message Transfer Sublayer Validation Test - ISIS File System
(continued)

```
>5 (title)
Device: DSK/:F1:TEST.TXT
Data: TEST OF TITLE ON ISIS FILE SYSTEM
Device Indicates: OK

>3 (write)
Device: DSK/:F1:TEST.TXT
Data: THIS IS A TEST OF THE ISIS FILE SYSTEM
THIS IS A TEST OF THE ISIS FILE SYSTEM      [DSK/:F1:TEST.TXT]
                    (verified by using ISIS utilities)
Device Indicates: OK                                  [CON]

>1 (close)
Device: DSK/:F1:TEST.TXT
Device Indicates: OK

>0 (open)
Device: DSK/:F1:TEST.TXT
Device Indicates: OK

>2 (read)
Device: DSK/:F1:TEST.TXT
Data:  P ("<ctrl>-@ P" was the two character sequence
          entered to indicated a buffer length of 80)
THIS IS A TEST OF THE ISIS FILE SYSTEM
Device Indicates: END-OF-FILE

>2 (read)
Device: DSK/:F1:TEST.TXT
Data:  P ("<ctrl>-@ P" was the two character sequence
          entered to indicated a buffer length of 80)
Device Indicates: END-OF-FILE

>1 (close)
Device: DSK/:F1:TEST.TXT
Device Indicates: OK

>7 (rename)
Device: DSK/:F1:TEST.TXT
Data: :F1:NEW.TXT
Device Indicates: OK

>6 (delete)
Device: DSK/:F1:NEW.TXT
Device Indicates: OK

>8 (reset)
Device: DSK
Device Indicates: OK
```

```
>2 (test)
Device: DSK
Device Indicates: OK

>G (get config)
Device: DSK
Device Indicates: Invalid Command

>S (set config)
Device: DSK
Data: 00000000
Device Indicates: Invalid Command


(Series III Console Test) ----------------------------------

>0 (open)                                                [CON]
Device: CON
Device Indicates: OK

>1 (close)
Device: CON
Device Indicates: OK

>2 (read)
Device: CON
Data:  P ("<ctrl>-@ P" was the two character sequence
                entered to indicated a buffer length of 80)
THIS IS A TEST OF READING THE CONSOLE
THIS IS A TEST OF READING THE CONSOLE
Device Indicates: OK

>3 (write)
Device: CON
Data: THIS IS A TEST OF WRITING TO THE CONSOLE
THIS IS A TEST OF WRITING TO THE CONSOLE
Device Indicates: OK

>4 (page)
Device: CON
(the screen cleared and the cursor appeared in the upper left)
Device Indicates: OK

>5 (title)
Device: CON
(the screen cleared and the cursor appeared in the upper left)
Device Indicates: OK
```

Message Transfer Sublayer Validation Test - Series III Console
(continued)

>6 (delete)
Device: CON
Device Indicates: Invalid Command

>7 (rename)
Device: CON
Data: PPP
Device Indicates: Invalid Command

>8 (reset)
Device: CON
Device Indicates: OK

>9 (test)
Device: CON
Device Indicates: OK

>G (get config)
Device: CON
Device Indicates: Invalid Command

>S (set config)
Device: CON
Data: 00000000
Device Indicates: OK

(End of User Agent Sublayer Validation Test)

III. System Integration Test

This test was not performed due to the lack of an Interface Processor Board for the second Series III MDS AP system. Also, without a hard disk system, there is not sufficient storage on the Debugger system for the executable code module.

Vita

First Lieutenant Kenneth N. Cole was born on 21 September 1949 in Highland Park, Michigan. He graduated from Kimball High School in Royal Oak, Michigan, in 1967 and entered an engineering program at the University of Michigan. In 1970, he enlisted in the U.S. Air Force and served as an Automatic Flight Control Systems Technician (AFSC 32570) at R.A.F Bentwaters, England. During that time, he earned a Bachelor of Science degree in Business Management from the University of Maryland overseas program.

In 1977, then Staff Sergeant Cole was assigned to Myrtle Beach A.F.B., South Carolina, and in 1978 he was selected to attend the University of Florida under the Airman Education and Commissioning Program (AECP). He received a Bachelor of Science degree in Electrical Engineering in 1980 and was commissioned later that year. His initial assignment was to the Aeronautical Systems Division, AFSC, at Wright-Patterson A.F.B, Ohio.

Lieutenant Cole entered the Air Force Institute of Technology in June 1982.

Permanent Address: 1009 Middy Drive
Wright-Patterson A.F.B, Ohio 45433

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GE/EE/83D-17	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable): AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. — PROGRAM ELEMENT NO.; PROJECT NO.; TASK NO.; WORK UNIT NO.
11. TITLE (Include Security Classification): See Box 19	
12. PERSONAL AUTHOR(S): Kenneth N. Cole, B.S.E.E, 1st Lt, USAF	
13a. TYPE OF REPORT: MS Thesis; 13b. TIME COVERED: FROM _____ TO _____; 14. DATE OF REPORT (Yr., Mo., Day): 1983 December; 15. PAGE COUNT: 303	
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES — FIELD: 9; GROUP: 2; SUB. GR.:	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Communications Networks, Input Output Processing, Multiprocessors, Microprocessors

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: DESIGN AND IMPLEMENTATION OF AN INPUT/OUTPUT INTERFACE PROTOCOL FOR THE INTEL 432/670 COMPUTER SYSTEM

Thesis Chairman: Dr. Gary B. Lamont

Approved for public release: IAW AFR 190-17.
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Gary B. Lamont	22b. TELEPHONE NUMBER (Include Area Code): 513-255-3576 — 22c. OFFICE SYMBOL: AFIT/ENG

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE

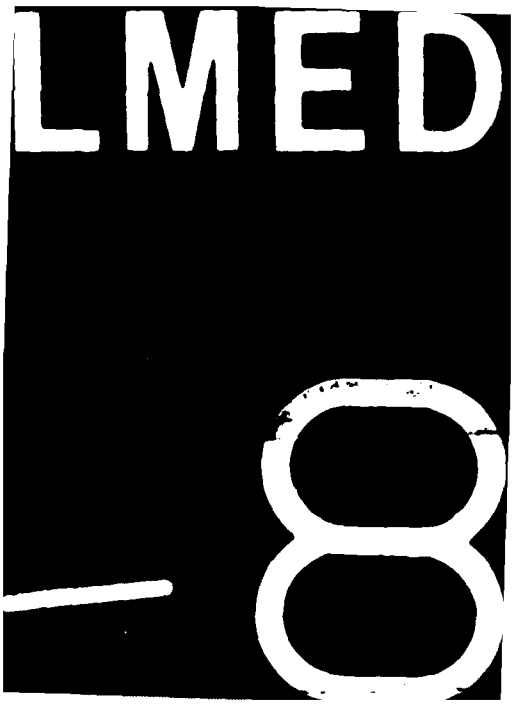
Block 19 (continued):

Abstract:

Distributed computer systems have many advantages to offer in terms of simplicity, efficiency, protection, and security as well as improved performance from the concurrency in such a system. Communication among distributed processors is a key issue in the design of a distributed system. While the lower levels of a communication system are generally defined by the hardware configuration and thus, implementation dependent, protocols for communication may be developed at higher levels that are independent of the hardware implementation. Using the Computer Based Message System under development for the National Bureau of Standards, this thesis investigation is an attempt to develop a usable I/O interface for a distributed computer system.

The Intel 432 Micromainframe computer system is a functionally distributed multiprocessor system. The hardware organization and operating system features lend themselves to the development of a message based communication system among users and devices on distinct processors.

This specific research effort involves defining the protocol requirements, as well as designing, implementing, and testing a distributed I/O system communication interface on the 432 computer system.


LMED
8